Proceedings of the Workshop on Automatic Data Layout and Performance Prediction

Rice University
April 19–21, 1995

CRPC-TR95548

June, 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
Houston, TX 77005-1892

Forward

The workshop on Automatic Data Layout and Performance Prediction (AP'95) brought together some of today's leading researchers in the field of automatic data layout and performance estimation. Thirty-one researchers from eight countries participated in the workshop. Participants presented their work through talks and demonstrations. We solicited papers and technical reports from each participant. The submitted papers and reports cover the most recent work of each researcher or research group.

This document contains the abstract and submitted paper for each presentation. The abstracts, submitted papers, and submitted slides of the presentations are also available through the World Wide Web at the following URL:

http://www.cs.rice.edu/~kremer/AP95

This workshop was a follow-up to the International Workshop on Automatic Parallelization (AP'93), which was held in Saarbruecken, Germany, in March 1993. Approximately twenty researchers from the United States, Great Britain, Austria, and Germany took part in AP'93.

We would like to thank Vikram Adve, Debbie Campbell, Theresa Chatman, Dionne Williams, and Jacqueline Williams for their help in organizing the workshop. In particular, Vikram was in charge of the panel session on performance estimation and Theresa took care of most of the organizational challenges that come with hosting a workshop. Without their efforts this workshop would not have been such a success.

If you have any questions regarding this workshop, please feel free to contact Uli Kremer (kremer@cs.rice.edu).

This workshop was partially supported by the Center for Research on Parallel Computation, a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008, and by ARPA under contract number DABT63-92-C-0038. The content of this document does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Uli Kremer and Ken Kennedy
Rice University
Center for Research on Parallel Computation

Workshop Program

Wednesday, April 19

Session 1
8:35am - 10:05am

Demonstration of Automatic Data and Computation Decomposition Techniques
Jennifer Anderson, Stanford University

Automatic Data Transformation for Shared Address-Space Machines
Saman Amarasinghe, Stanford University

An Affine Loop Transformation Algorithm for Parallelism
Amy Lim, Stanford University

Session 2
10:30am - 12:00pm

The Subspace Model: A Theory of Shapes for Parallel Systems
Kathleen Knobe, MIT
William J. Dally, MIT

Efficient Distribution Analysis Via Graph Contraction
Tom Sheffler, RIACS
Rob Schreiber, RIACS
Bill Pugh, University of Maryland
John Gilbert, Xerox PARC
Siddhartha Chatterjee, University of North Carolina

Automatic Data Mapping And Program Transformations
Jagannathan Ramanujam , Louisiana State University

Session 3
1:30pm - 3:00pm

A Novel Approach Towards Automatic Data Distribution
Jordi Garcia, Universitat Politecnica de Catalunya, Spain
Eduard Ayguade, Universitat Politecnica de Catalunya, Spain
Jesus Labarta, Universitat Politecnica de Catalunya, Spain

Automatic Data Layout for High Performance Fortran
Ulrich Kremer, Rice University
Ken Kennedy, Rice University

Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers
Daniel J. Palermo, University of Illinois at Urbana-Champaign
Prithviraj Banerjee, University of Illinois at Urbana-Champaign

Session 4
3:30pm - 4:30pm

Pattern-Driven Automatic Parallelization
Christoph W. Kessler, University of Trier, Germany

Program Comprehension Techniques to Improve Automatic Parallelization
Beniamino Di Martino, University of Naples, Italy
Barbara Chapman, University of Vienna, Austria

Reception and Computer Demonstrations
5:00pm - 6:30pm

Computer Demonstrations:
Thomas Fahringer, University of Vienna, Austria
Jordi Garcia, Universitat Politecnica de Catalunya, Spain
Ulrich Kremer, Rice University
Harikumar Sivaraman, Washington State University

Thursday, April 20

Session 5
8:30am - 9:30am

ADDT: Automatic Data Distribution Tool for Porting Programs to Distributed Memory Machines
Harikumar Sivaraman, Washington State University
Cauligis Raghavendra, Washington State University

Automatic Mapping of Task and Data Parallel Programs
Jaspal Subhlok, Carnegie Mellon University

Panel Session on Automatic Data Layout
9:45am - 10:45am

Panelists:
Guang R. Gao, McGill University, Canada
Manish Gupta, IBM T.J. Watson Research Center
Jagannathan Ramanujam, Louisiana State University
Rob Schreiber, RIACS
Jaspal Subhlok, Carnegie Mellon University

Session 6
11:00am - 12:00pm

P3T: An Automatic Performance Estimator for Parallel Programs
Thomas Fahringer, University of Vienna, Austria

Locality Analysis and Optimization for SMPs
Guang R. Gao, McGill University, Canada
Vivek Sarkar, IBM Software Solutions Division

Session 7
1:30pm - 2:30pm

Static Estimation of Communication Costs of Data-Parallel Programs
Manish Gupta, IBM T.J. Watson Research Center
Prithviraj Banerjee, University of Illinois at Urbana-Champaign

Performance Prediction and Scalability Analysis for Data Parallel Programs
Celso L. Mendes, University of Illinois at Urbana-Champaign
Jhy-Chun Wang, University of Illinois at Urbana-Champaign
Dan Reed, University of Illinois at Urbana-Champaign

Panel Session on Performance Prediction
2:45pm - 3:45pm

Panelists:
Thomas Fahringer, University of Vienna, Austria
Arjan J.C. van Gemund Technical University Delft, The Netherlands
Dan Reed, University of Illinois at Urbana-Champaign
Vivek Sarkar, IBM Software Solutions Division

Friday, April 21

Session 8
8:30am - 9:30am

A Cost Model and Flow Analysis Based Heuristic for Automatic Data Partitioning
Jodi Tims, University of Pittsburgh

Predicting Contention in Distributed-Memory Machines
Arjan J.C. van Gemund, Technical University Delft, The Netherlands

Session 9
9:45am - 10:45am

Analyzing the Behavior and Performance of Parallel Programs
Vikram Adve, Rice University

Dynamic Performance Prediction within the ESPRIT PPPE Project
Alistair Dunlop, The University of Southampton, Great Britain
(abstract)

Wrap Up
10:45am - 11:15am

Abstracts and Papers

Demonstration of Automatic Data and Computation Decomposition Techniques

Jennifer Anderson, Stanford University

This talk describes and evaluates a compiler algorithm that automatically finds computation and data decompositions that optimize both parallelism and locality. The algorithm is based on a linear algebra framework that enables us to systematically derive the decompositions. Both the alignment and distribution of the computation and data are calculated simultaneously.

Our algorithm can exploit parallelism in both fully parallelizable loops as well as loops that require explicit synchronization. The algorithm will trade off extra degrees of parallelism to eliminate communication. If communication is needed, the algorithm will try to introduce the least expensive forms of communication into those parts of the program that are least frequently executed.

We implemented our algorithm as part of the SUIF compiler system, and tested it on application programs from the Perfect, SPECFP and NAS benchmark suites. In this talk we present various statistics about the decompositions the algorithm finds. We also ran the compiler generated code for a number of the applications on the Stanford DASH multiprocessor. The experimental results demonstrate that our optimizations can greatly improve program performance.

Automatic Data Transformation for Shared Address-Space Machines

Saman Amarasinghe, Stanford University

Data decompositions, either automatically generated by parallelizing compilers or manually provided by HPF programmers, specify which data are local to each processor in a distributed address space machine. While data decompositions are unnecessary for shared address space machines, such information can also be used effectively to change the data layout on these machines so as to improve spatial locality, minimize false sharing and decrease cache conflicts.

This talk describes a set of transformations to restructure the memory layout of distributed arrays on shared address space machines. The framework used for performing data transformations is analogous to the theory of loop transformations. We present an algorithm that derives the desired transformations from data decompositions. This talk also includes some experimental results based on our implementation of the techniques in the Stanford SUIF compiler system. The results suggest that data transforms are very important to the overall performance of shared address space machines.

An Affine Loop Transformation Algorithm for Parallelism

Amy Lim, Stanford University

This talk presents a parallelization algorithm that combines unimodular transformations with loop fusion, fission, scaling, reindexing and statement reordering to optimize programs consisting of arbitrary nestings of loops and sequences of loops. Our algorithm first finds the maximum amount of parallelism available given that no communication is allowed. Only when the parallelism available is found to be insufficient do we introduce communication and synchronization in the least frequently executed parts of the computation to obtain more parallelism. To do so, the algorithm partitions the computation into stages, with each stage constituting an iteration of a sequential loop. Our algorithm derives the partition in such a way that computation within each stage has as much parallelism as possible. Once the computation is partitioned into sequential stages, we parallelize the computation in each stage using the same parallelization algorithm.

The Subspace Model: A Theory of Shapes for Parallel Systems

Kathleen Knobe and William J. Dally, MIT

Although there are many sophisticated algorithms for data layout and scheduling, the input to these algorithms, that is the objects to be laid out and the code fragments to be scheduled, are typically taken to be the data structures, as specified by the user and the loop iterations, as specified by the user. This paper presents an analysis of programs that improves the results of existing layout and scheduling algorithms by improving the input to these algorithms.

The work is based on a theory of shapes (subspaces of the iteration space) that generalizes existing shape related optimizations and strategies. We claim that there is some confusion in the literature between shape and location and, further, that the wrong shape can be more costly than the wrong location.

The paper presents the subspace abstraction and its related analyses and optimizations.

The Subspace Model: A Theory of Shapes for Parallel Systems*

Kathleen Knobe[‡†] William J. Dally[‡]

April 3, 1995

Artificial Intelligence Lab., Massachusetts Institute of Technology,
545 Technology Square, Cambridge, 02139, U.S.A.

Abstract

This paper presents a shape based abstraction. Data layout is often the subject of direct analysis while shape is addressed in ad hoc ways at best. However, a suboptimal shape can be more costly than a suboptimal location. Unnecessary serialization can result if the shape used is too small and unnecessary communication and computation can result if the shape is too large. For each dimension in the space of an object, the object attains that dimension, serially, in parallel or via a parallel prefix operations. These expansion categories are O(N), O(1) and O(log N) respectively, where N is the extent of the dimension. Using an expansion category slower than the natural one is unnecessarily slow.

There are three major impacts of this work.

First, the subspace model is useful by itself. The subspace abstraction subsumes existing shape related optimizations (such as privatization and invariant code motion) and shape related strategies (such as owner computes and scalar replication). In this capacity it removes excess communication, excess computation and unnecessary serialization.

Second, the subspace model has an impact on the requirements for earlier phases. The processing to determine natural shapes and natural expansion categories uncovers parallelism. This implies that the source code need not be parallel.

Third, the subspace model has an impact on the effectiveness of subsequent phases. By converting the code to natural shapes and natural expansion categories, it improves the input to the data partitioner and the scheduler by providing them with additional flexibility.

1 Introduction

Although there are many sophisticated algorithms for data partitioning and scheduling, [1, 2, 3, 4, 10, 11, 13, 16] the inputs to these algorithms, that is the objects to be partitioned and the code fragments to be scheduled, are usually determined naively. Generally the objects to be partitioned are the named arrays as declared by the user and the code to be scheduled

*A version of this paper will also appear in the Fifth Workshop on Compilers for Parallel Computers Malaga, Spain. June 1995.

†Supported in part by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

‡Supported in part by the Air Force Systems-ESD under contract F19628-92-C-0045.

is based on loop iterations as specified by the user.[1] The goal of this work is to improve the results of data layout and scheduling, not by improving the data layout and scheduling algorithms themselves, but by altering the input to the algorithms. This view has lead to an abstraction based on subspaces and a class of optimizations within that abstraction.

A shape is a subspace of the iteration space. We will use the notation $\{i, j\}$ to refer to the subspace spanned by the indices (vectors) `i` and `j`. Subspace is distinct from dimensionality. For example, $\{i, j\}$ and $\{i, k\}$ are distinct subspaces but of the same dimensionality. Notice also that that subspace is distinct from that of location. Two objects can be in the same subspace, say $\{i, j\}$, but in distinct locations and two objects can be in distinct subspaces, say $\{i, j\}$ and $\{i, k\}$, but in the same location. The subspace abstraction is also distinct from the virtual processor abstraction used during alignment in that, for example, `a(i)` and `a(i+1)` are in distinct virtual processors but are in the same subspace, $\{i\}$.

The examples below show that the user can create objects whose dimensionality is too small or too large.

Consider the two statements below with the same right hand sides (RHS) but distinct left hand sides (LHS). Assume that they are each within loops on `i`, `j` and `k`.

```
s1: a(i, j, k) = b(k) * c(i)
s2:       a(i) = b(k) * c(i)
```

If the reference `b(k)` refers to a 1-dimensional object with values at points along the `k` axis and the reference `c(i)` refers to a 1-dimensional object with values at points along the `i` axis, then their product, `b(k) * c(i)`, is a 2-dimensional object with values at points in the `ik` plane. In `s1` the LHS has more dimensions than the LHS expression. In `s2` the LHS has fewer. Inefficiencies may result if the subspace is not just right.

In statement `s1` since we assign this 2-dimensional object to the 3-dimensional array `a(i,j,k)`, we may incur the cost of unnecessary communication to deliver those values to each point along the `j` axis. Worse yet, if we employ the *owner computes* rule (described below), `b` will be communicated across `i` and `j` axes and `c` will be communicated across `j` and `k` axes. Since only the 2-dimensional result needs to be expanded across `j` this communication is excessive. In addition, the number of multiplies is $imax * jmax * kmax$, a factor of $jmax$ more than required. In general, if a subspace has too many dimensions, we may incur the cost of unnecessary communication and unnecessary computation.

Although statements like `s1` are not very common, the owner computes rule applied to a similar statement `S3`

```
s3:  a(i, j, k) = x(i, j, k) + b(k) * c(i)
```

would cause the identical problem.

On the other hand, in statement `s2`, since we assign this same 2-dimensional object to the 1-dimensional array `a(i)`, we must serialize along the missing axis, `k` in this case. Also, all uses of `a(i)` must be completed at one point along `k` before `a(i)` can be defined at the next point along `k` (i.e., anti-dependences must be observed). In addition, the execution order of all assignments to `a(i)` for the same value of `i` must be maintained (i.e., output-dependences must be observed). In general, if the subspace of the LHS object has too few dimensions, we may incur the cost of unnecessary serialization.

The examples above demonstrate that the appearance of an index as a subscript in the source does not imply that the index is part of the subspace and the absence of an index as a subscript in the source does not imply the absence of the index from the subspace.

Although there are many compilation strategies and optimizations whose goal is to affect the shape of objects or operations, they are each aimed at distinct effects and operate in

[1] Although the grouping of iterations into chunks in order to optimize for granularity and aspect ratio, is often determined automatically, the individual iteration is usually that given by the user.

specific and restricted ways. Some examples of shape affecting compiler strategies are listed below. These are taken from the Single Program Multiple Data (SPMD) paradigm used by High Performance Fortran (HPF) [9].

- the owner computes rule - specifies the shape of all the *operations* in an expression to be the shape of the left hand side array section. It does this indirectly by specifying that the location of the computations is the location of the LHS.[2] This rule makes no distinction between shape and location.
- scalars are replicated - specifies that all *named objects* that are *scalars* will have a location on each processor. In other words, the shape of a scalar is the shape of the processor configuration. This rule applies only to named scalar objects. It does not apply to small arrays, to dimensions of arrays whose extent is small or to expressions whose value is a scalar.
- control flow expressions are replicated - specifies that the value of any *expression* that is *used as a predicate* will be distributed to all processors. In other words, the shape of such an expression is the shape of the expression itself expanded by the shape of the configuration.

In addition to the compiler strategies that affect shape mentioned above, we list below some optimizations designed to affect shape.

- privatization - is a transformation that *adds* a loop index to the shape of a *named object* in the specific case when all references to that object within an iteration of the loop refer to the value defined within that same iteration.
- invariant code motion - is a transformation that *removes* a loop index from the shape of an *operation* when the results of all iterations are identical.

As is evident by the examples above, the goals and domain of each of these strategies and optimizations are restricted in somewhat arbitrary ways. Some apply only to named objects, others only to expressions. Some distinguish between scalars and arrays. Privatization has the specific goal of increasing the dimensionality whereas invariant code motion has the specific goal of decreasing the dimensionality. On the other hand, sometimes no distinction is made where one might prove useful. For example, the owner computes rule does not distinguish between shape and location even though the cost of reshaping might be greater than the cost of relocating.

The goal of this work is to develop a general theory of shapes that subsumes or improves on the compilation strategies and optimizations above, that automatically determines the objects to be distributed and the code to be scheduled and that supports a variety of parallel targets.

The philosophy is analyze first, transform, then synthesize, rather than successive transformations, each of which results in legal source code. The analysis phase first converts the code into single assignment form, then determines the natural subspace of the object associated with each assignment (Sections 2 and 3). The result of the combination of static single assignment and natural subspaces is a form that is dynamic single assignment at the element level, that is each element is assigned to at most once during execution. Finally the analysis phase determines the natural expansion category of each index in the subspace of each object (Section 5). Natural expansion category determination distinguishes between objects attaining an axis via a serial loop, via a parallel-prefix loop and via a parallel loop. The synthesis phase generates the appropriate set of loops, serial, parallel or scans as indicated by the expansion category (Section 6). This process may break up code that was

[2]There have been efforts to improve on the owner computes rule [4, 10] but they are within the domain of data partitioning.

within a single iteration into several distinct loops with possibly distinct expansion categories. The advantages of the resulting code as input to the data partitioner and scheduler are addressed in (Section 7). The optional transformation phase, if included, occurs between the analysis and synthesis phases. It optimizes expressions given the subspaces and expansions of the leaves of the expression trees (Section 8).

2 Natural Subspaces

This section introduces some relevant concepts and terminology. We will refer to the statement below which, like most of the examples in this paper, is scalar code. We will be uncovering additional dimensionalities of objects and determining the natural type of parallelism of each dimension of each object defined. If some of the dimensions are already specified via declarations and some of the parallelism is specified, for example, via vector operation notation in Fortran90, these are incorporated directly into the algorithms; however, since the interesting aspects of our approach occur when a dimension or the parallelism is not explicit, most of the examples are simply scalar.

Assume `S4` is within loops on `i` and `j`.

```
s4:  c(i) = d(i+1, j) * e(i) + i
```

The term *named object* denotes any textual occurrence (definition or reference) of a scalar or array that is given a name in the source (e.g., `c` in `S4`). *Unnamed objects* are the results of computations (e.g., `d(i+1, j) * e(i)` in `S4`) . An *object* is either a named or an unnamed object. An *element* of an n-dimensional object is specified by `n` subscripts, one associated with each dimension. Each such element has a single value. An *index* is a loop index. The *natural subspace* of an object is a *subspace* of the iteration space, such that an index is part of the natural subspace of an object if the value of an element referenced may vary as the index varies. In general, the subspace of an object (named or unnamed) is the span of the subspaces of the objects used to compute it. If the subspace of `d(i+1, j)` and `e(i)` are $\{i, j\}$ and $\{i\}$ respectively, then the subspace of their product is $\{i, j\}$.

From the perspective of the source program, the value of a reference may change for two reasons, space or time. The element referenced may change as an index that appears in a subscript expression varies. For example, as `i` varies the value of `e(i)` varies because a distinct element is referenced. This is a change in space. However, even if the index does not appear as a subscript, it is possible that the value changes. For example, as `j` varies the reference to `e(i)` may vary if there is an assignment to `e` within in the `j` loop. This change does not correspond to a change in space, however, since the same element is referenced. The change is based on a change of time as the `j` loop is executed. In general, changes over space provide more flexibility than changes over time. In the above example, as `i` changes over space, and `e(i)` is defined over the range of `i`, all the values are available. However, for a given `i`, as `j` changes over time, and `e(i)` is defined for a given `i` over the range of `j`, only the most recent value is available. We will be converting the source with its changes over both space and time to an internal form called subspace internal form that represents all changes as changes over space. See [15] for a discussion of data structures relating time, data space and processor space.

Consider the space $\mathbf{Z}^n$ with basis vector $\{e_1, e_2, ...e_n\}$ where each e_i is an index in the iteration space. This set has 2^n subsets. Each subset spans a subspace of the iteration space. For example, objects defined within loops on `i`, `j` and `k` may be in one of the eight distinct subspaces of the iteration space corresponding geometrically to the origin of the cube ($\{\}$), the axes of the cube ($\{i\}$, $\{j\}$, $\{k\}$), the faces of the cube ($\{i, j\}$, $\{i, k\}$, $\{j, k\}$) and the whole cube ($\{i, j, k\}$). Notice that this abstraction distinguishes among, for example, the 3 different 2-dimensional planes that have different orientations but does not distinguish among different positions on a given parallel plane, that is, it ignore details of expressions

in subscripts. For example, regardless of whether the reference is `c(i)`, `c(i+1)`, `c(i*2)`, `c(sin(i))`, `c(v(i))`, or even `c(k)`, `i` is in the subspace of the reference if and only if the value of the reference may change as `i` varies.

The *dimensionality* of an object is the cardinality of its natural subspace. (If the natural subspace of the unnamed object `d(i+1, j) * e(i)` is $\{i, j\}$, its dimensionality is 2.) A loop index is always a 1-dimensional object (The natural subspace of `i`, used as a term in `S4`, is $\{i\}$). Notice that we distinguish between constants (known to be in the empty subspace), loop indices (known to be in a 1-dimensional subspace) and all other references. We do not distinguish between array and scalar references (other than loop indices) because the algorithm will uncover the natural subspace of an object whether it starts out with 0, 1, or more dimensions.

A *definition* is an occurrence of a name that modifies values. A *reference* is an occurrence of a name that does not modify values. Consider an assignment to `x(i,k)` that creates a 2-dimensional object in $\{i, k\}$ space. A subsequent reference to `x` may refer to `x(i,k)`, also 2-dimensional and also in $\{i, k\}$ space. It may refer to `x(i,i)`, a 1-dimensional reference in $\{i\}$ space. It might refer to `x(i,v(k,j))`, a 3-dimensional reference in $\{i, j, k\}$ space.

Since each assignment may define an object of distinct subspace, several assignments to the same name in the source may define distinct objects in the subspace internal form specifically so the partitioner and scheduler can treat them independently.

A dimension in a reference is a *contributing dimension* if it contributes to the subspace. Consider the reference `x(i, 3)` where `i` is a loop index. The second dimension can not be a contributing dimension. If `i` belongs to the subspace of the reference, that is, if the value of `x(i, 3)` varies as `i` varies, then the first dimension is a contributing dimension. How we determine if `i` is part of the subspace is the subject of Section 3.

In the conversion to objects in internal subspace form, dimensions of the source object are either eliminated or maintained. Reexamine the assignment `s1` in the previous section. Since the reference `a(i, j, k)` is determined to be in subspace $\{i, k\}$, the first and third dimensions of the source reference are considered *maintained dimensions*. The second dimension of the source reference will not be represented in the subspace object so it is considered an *eliminated dimension*. We determine a *reference mapping* so that given a source reference and the iteration indices, we can generate a reference to the internal subspace object. We will use a reference mapping notation $< (i, ., k) >$ in this case, to indicate that the first dimension is maintained and contributes `i`, the second is eliminated and contributes nothing and the third is maintained and contributes `k`.

Dimensions of the subspace object generated are either source dimensions or expansion dimensions. Reexamine the assignment `s2` in the previous section. Since the source reference `a(i)` is determined to be in subspace $\{i, k\}$, the internal object generated is 2-dimensional with reference mapping $< (i)[k] >$[3] indicating that the first of the two dimensions in the internal object is a source dimension associated with index `i`, whereas the second is an expansion dimension associated with index `k`. The reference mapping notation may be nested when array subscripts are themselves arrays.

Recall that the subspace of the result of an arithmetic operation is the span of the subspaces of its arguments. An array reference is an operation whose arguments are the objects created by the subscript expressions of the contributing and expansion dimensions. The subspace of the result of an array reference is the span of the subspaces of these arguments. The determination of natural subspaces then involves determining the contributing and expansion dimensions for each array reference.

[3] Multiple expansion indices are ordered canonically from outer to inner loops in this notation.

3 Algorithm for Determining Natural Subspaces

Now we are ready to present an initial subspace determination algorithm. For ease of understanding, we first consider only the language consisting of assignments and `do` loops. This restriction is relaxed in Section 9. This algorithm will determine a subspace for each textual occurrence of a named object.

Briefly, after an initialization phase (See Section 3.1), the algorithm propagates indices from the RHS to determine the subspace of the LHS. The indices of the subspace of the LHS determine the contributing dimensions of the LHS object, that is, the reference mapping (See Section 3.2). The contributing dimensions of the LHS object determine the subspace of the RHS references to that object (See Section 3.3). This propagation continues until there are no further changes. The algorithm is executed in worklist style. A reference is inserted onto the worklist when its current state changes. A references is pulled off the worklist to propagate its change. No particular order of processing, such as inner loops first, is necessary.

The reference mapping described earlier showed the end product of this analysis. During the propagation, however, two temporary situations may hold. `?` indicates that, for some dimension, we do not yet know if it contributes or not. `*` indicates that we know the dimension contributes but we do not yet know what indices it contributes.

Details of the algorithm follow.

3.1 Initialization

- Constants are in subspace ϕ. They do not vary with any index.
- Indices are in their own subspace. Consider the expression `a(i, j) + j`, in loops on `i` and `j`. The subscripts `i` and `j` and the term `j` are all known initially to be 1-dimensional in subspaces $\{i\}$, $\{j\}$ and $\{j\}$ respectively.
- For each reference that is live on entry to the routine (arguments and globals), each dimension is known initially to be a contributing dimension. A reference `a(i, j)` that is live on entry and is within loops `i` and `j` is initially known to be a 2-dimensional reference in $\{i, j\}$ with reference mapping $< (i, j) >$.

  Note that if a source array is referenced several times, some of the references may be live on entry while others are not. Only the references that are live on entry are set by initialization. The subspaces of references not live on entry are determined by the algorithm.

  If, in the reference `a(i, j)` that is live on entry, `i` is not a loop index but rather a user defined scalar, the first dimension of the reference above is known to contribute but until we determine the subspace of `i` we don't know what indices it actually contributes. In this case, its reference mapping is $< (*, j) >$.

3.2 Propagate from the RHS to the LHS

Subspace: If an index, `i`, is part of the subspace of one of the references on the RHS of an assignment, then it is part of the subspace of the LHS object. In the expression `d(i+1, j) * e(i)`, when we determine that `j` is in the subspace of `d`, `j` will be added to the subspace of the LHS object and when we determine that `i` is in the subspace of `d` or `e`, `i` will be added to the subspace of the LHS object. The LHS object will be in natural subspace $\{i, j\}$ upon termination regardless of the number of dimensions declared for the LHS in the source.

Reference mapping: The mapping of an LHS to internal subspace form is determined by the indices in its subspace. Consider an LHS, `a(i, j)`. When index `i` is added to the subspace, the first dimension becomes a contributing dimension with reference mapping $< (i, ?) >$. When index `k` is added to the subspace, an expansion dimension is added and

the reference mapping becomes $< (i, ?)[k] >$. If the algorithm terminates without adding `j`, the final reference mapping for this reference is $< (i, .)[k] >$

3.3 Propagate from the LHS to each RHS reference

- For each contributing dimension in a source position of an LHS

  Subspace: If subscript position p of an LHS reference is contributing then position p of any RHS reference it reaches is contributing. Consider the following code within loops on `i`, `j` and `k`:

```
s5:   a(i, j, k) = s * b(i, j)
s6:        . . . = a(i, j, k)
s7:        . . . = a(i, k, j)
```

  If LHS of `s5` is in space $\{i, k\}$, the first and third dimensions are contributing. This definition reaches RHS references in `s6` and `s7`. Therefore, in both cases, the first and third dimensions of these references are also contributing. A position, p, in a single reference reached by multiple definitions, is contributing if position, p, is contributing in any of these definitions.

  Reference mapping: If position, p, in an RHS reference is contributing, then the index on which the subscript in position, p, is based is part of the subspace of that reference. In both of the two references above, `s6` and `s7`, the first and third dimensions are contributing. The reference mappings are therefore $< (i, ., k) >$ and $< (i, ., j) >$ respectively. The first reference is in subspace $\{i, k\}$ whereas the second is in subspace $\{i, j\}$ even though both came from a definition in $\{i, k\}$.

- For each expansion dimension of an LHS

  Subspace: Consider the following example.

```
s8:  a = 0
     do i = 1, imax
s9:      a = a + b(i)
s10:      ... = a ...
     enddo
s11:  ... = a ...
```

  The LHS reference to `a` in `s9` has reference mapping $< ()[i] >$, with expansion index, `i`. This reference reaches three RHS references, in `s9`, `s10` and `s11`. Since the first two references are within the scope of `i`, `i` is included in each of their subspaces as an expansion index.[4] However, since the reference in `s11` is not within the loop on `i`, `i` is not included. A new object is created by *reduction* along that axis. Reduction typically involves taking the last value along an axis to create a new lower dimensional object.

  Reference mapping: The reference mapping for the references in `s9` and `s10` are $< ()[i] >$ but the reference mapping for the reference in tt s11 is $< () >$.

To assess the complexity of the algorithm, consider the processing of the true dependence on `x` in the following code within loops on `i`, `j` and `k`

```
s12: x = b + c * d
 ...
s13:   = ...   x
```

[4]Since the reference in `s9` is reached via a loop carried dependence, the actual expansion will refer to the previous element along that axis, that is it will become `a(i) = a(i-1) + b(i)`.

If at some point, during the propagation, $\{i\}$ is added to the subspace of one of the RHS references in `s12` then `i` is propagated to the subspace of `x` on the LHS. If the subspace of `x` did not already include `i` then `i` is propagated through the dependence to the reference in `s13`. At some later time, if $\{j\}$ is added to the subspace of the same or another RHS reference in `s12` then `j` is propagated to the subspace of `x` on the LHS. The same dependence to the reference in `s13` is then processed a second time. It may be processed a third time for index `k`. It can be processed at most once for each enclosing loop. Consider processing of a dependence to include both the propagation required from an LHS to an RHS and form that RHS to its own LHS. In the worst case each true dependence is processed once for each loop index. This algorithm is $O(D * L)$ where D is the number of true dependences (Anti- and output-dependences are ignored.) and L is the maximum loop depth.

Natural subspace analysis determines a set of objects and their subspaces. The objects include the named objects as determined above and the unnamed objects whose subspace is the span of the subspaces of their operands.

4 Operational Subspace

On distributed memory systems, operations occur between two operands of the same subspace. Corresponding elements are combined to produce a result. Assuming the obvious natural subspaces, the assignment `a(i,j) = b(i) + c(i,j)` in loops on `i` and `j` requires that we expand `b` from its natural 1-dimensional subspace, $\{i\}$, to a 2-dimensional subspace $\{i, j\}$ before performing the operation. The *operational* subspace of an operand is the subspace of the operation in which it is involved. In order to participate in the operation, the operand must be in its operational subspace. Since the subspace of an operation is the span of the subspaces of its operands, natural subspace is always a subspace of the operational subspace. The expansion from natural subspace to operational subspace can be performed on a grid via replication in logarithmic time.

Notice that some expressions can be computed locally without cost in any space that includes than their natural space. Although a constant is in the empty subspace, it is available in any subspace without communication cost. In the expression `a(i, j) + i`, the term `i` is available in any subspace containing `i`, in particular in $\{i, j\}$, without communication. Such objects are referred to as *implicitly distributed* and incur no communication cost in expanding from their natural to their operational subspace.

One advantage to explicitly representing the operational subspace on a distributed memory system is that the expansion to an operational subspace may be redundant even when the specific communication is not. An example of this optimization is seen in the following examples. Assume that the code is within loops `i`, `j`, `k` and `m`.

```
s14:
 x(m,i,j,k)*v(i,j,k) + y(m,i,j,k)*v(i,j,k) + z(m,i,j,k)*v(i,j,k)
```

Assume `x`, `y` and `z` are all in subspace $\{m, i, j, k\}$ and assume that `v` is in subspace $\{i, j, k\}$. The communication to expand `v` to its operational subspace for each of the three multiplications will be exposed by the data layout phase. Assuming the obvious layout, with `x`, `y` and `z` aligned, two of the three expansions will be eliminated as redundant by optimizations after specific communication operations are exposed. However, consider the minor modification below.

```
s15:
 x(m,i,j,k)*v(i+1,j,k) + y(m,i,j,k)*v(i,j+1,k) + z(m,i,j,k)*v(i,j,k+1)
```

With the same layout of these objects, where the object `x`, `y` and `z` align, the actual communication to expand `v` across each of these objects will differ. So the expansion is not

redundant at the level of explicit communication based on a specific layout and there are no redundant operations in the source. However, if we look at the intermediate level of natural and operational subspaces, the expansion of `v` across the axis `m` is redundant for two of the three multiplications. At the subspace level, we expand `v` across `m` once and at the location level, we simply adjust the alignment of this expanded object by one for each of the subsequent operations. Notice that the same argument would apply if the indices of `v` were identical but either the subscripts or the layout of `z`, `y` and `x` were distinct. Although this example is within a single expression and might be handled by statement level communication optimization, the situation is more common between statements. (See Section 8.3.)

5 Natural Expansion Categories

When an array, `x`, is privatized [14, 17] along a loop index, `i`, two things happen. The array is given an additional dimension, in our terminology, an expansion dimension, and accesses to `x` along `i` are asserted to be parallelizable.

We extend this notion in two ways. First, we find not only parallel expansions but all expansions, determining the expansion category of each. Second, we determine this category, not only for expansion dimensions but for all dimensions of the internal object, expansion dimensions and source dimensions. These expansion categories are described below. The examples show both source and expansion dimensions.

- *Local* - the value of each element is not based (directly or indirectly) on elements earlier along that axis. The anticipated communication costs are O(1) for local axes. `s` in the following code is locally expanded across `i`.

```
do i = 1, imax                do i = 1, imax
 ...  = s                      ...  = s(i-1)
 s = x * y(i)                  s(i) = x * y(i)
 ...  = s                      ...  = s(i)
 enddo                        enddo
```

  Our definition of local is similar to that of privatization but differs in two important ways. First, as mentioned above, it applies to existing source dimensions as well as to expansion dimensions. In addition, it does not require that all references to an element be in the same loop iteration. In the first statement, the use of `s` defined on the previous iteration does not prevent the definition of `s` in the second statement from being local, though it would prevent the object from being privatized. The critical importance of privatization has been detailed. [7]

- *Linear* - the value of an element is potentially based on the value of elements earlier along that axis. The communication costs are O(N) where N is the extent of the axis. `s` in the following code is linearly expanded across `i`.

```
do i = 1, imax                do i = 1, imax
  s = userfunc(s)              s(i) = userfunc(s(i-1))
enddo                         enddo
```

  Here we must order the calls to `userfunc` as in the original code.

- *Scan* - the value of an element is based on the value of elements earlier along that axis but the operations can be converted to parallel-prefix form with communication cost of O(log N). `s` in the following code is scanned across `i`.

```
do i = 1, imax                    do i = 1, imax
   s = s + b(i)                    s(i) = s(i-1) + b(i)
enddo                             enddo
```

Just as the subspaces of intermediates are determined by the subspaces of their children, the expansion categories for the axes of intermediates are determined by the expansion categories of the axes of their children. Consider `a(i) = a(i-1) + b(i)`.[5] `a` is expanded via a scan across `i`. The object corresponding to the result of the `+`, is part of the scan because one of its children is part of the scan. In the assignment `a(i) = a(i-1) + b(i) * c(i)`, the `+` is still part of the scan but the `*` is not since neither child is part the scan.

The examples above are simplified in that in each, the axis in question corresponds to the innermost loop. In fact, an expansion category is determined for each dimension of an object, whatever level loop it corresponds to. In the following code, s is expanded via scan along `j` but is local along `i`.

```
do i = 1, imax
   s = 0
   do j = 1, jmax
      s = s + b(i, j)
   enddo
enddo
```

For linear and scan expansions, there is a *cycle* of nodes involving a loop carried dependence from an LHS to an RHS and from that RHS to the LHS within the same statement. The natural expansion category determination finds these cycles and distinguishes among distinct cycles created by loop carried dependences on the same loop. A given cycle may involve more than one object as in

```
do i = 1, imax
  x(i) = y(i) + ...
  y(i+1) = ...+ x(i)
enddo
```

The natural expansion category determination finds these distinct cycles and finds hierarchical relationships among cycles.

6 Restructuring

When the analysis component is complete, the natural subspace for each object, the natural expansion category for each axis of each object, and the cyclic relationships among expansions have all been determined. At this stage, a collection of optimizations available within the subspace abstraction constitute an optional transformation component (See Section 8). But before we discuss these optimizations, we should first understand how code is generated based on the analysis. This is the work of the synthesis component and is described in this section.

The subspace internal form will be transformed into hierarchically organized fragments based on subspaces, expansion categories and specific cycles, as determined by the analysis component.

An innermost fragment may be one of three types

- a *natural expansion fragment* of type local, linear, or scan (see Section 5)

[5] This might have appeared this way in the source or it might have been expanded to this form via a natural subspace determination on `a = a + b(i)`.

- a *reduction fragment* (see Section 3.3)
- an *operational expansion fragment* (see Section 4)

Innermost natural expansions are represented as individual fragments. For example, the three expansions above in Section 5 are represented as the three fragments below.

```
local (i = 1,imax)      linear (i = 1, imax)       scan (i = 1, imax)
 s(i) = x * y(i)         s(i) = userfunc(s(i-1))    s(i) = s(i-1) + b(i)
endfragment             endfragment                endfragment
```

Although these fragments include entire statements, in fact, the fragments are determined at the level of specific operations. Operations are in distinct innermost fragments if

- they are in distinct subspaces,
- they are in distinct expansion categories over a given index,
- they are in distinct cycles over a given index,
- or the expansion category for a given index is local for both but they define distinct objects.

Since a given cycle may contain parts of several statements and a given statement may not all be within the same cycles, a fragment may contain parts of several statements.

Consider the following code

```
y(i) = (a(i, j) + b(i)) + (x(i-1) + c(i))
x(i) = y(i) * d(i, j)
```

where the reference mappings for leaf nodes are

```
y<(i)[j]> = (a<(i,j)> + b<(i)>) + (x<(i)[j]> + c<(i)>)
x<(i)[j]> = y<(i)[j]> * d<(i,j)>
```

and the cycles for leaf nodes are:
(the cycle based on the loop carried dependence of `x` across the `i` axis is written as `x->i`)

```
{x->i} = ({} + {}) + ({x->i} + {})
{x->i} = {x->i} * {}
```

Breaking this code into fragments involves determining the subspace and the cycles for each intermediate node. All four operations are in $\{i, j\}$ space. Only the left most plus is not within the cycle. The following four fragments are generated. `F2` and `F4` are natural expansion fragments. `F1` and `F3` are operational expansion fragments. (There are no reduction fragments in this example.):

```
F1: replicate (j = 1,jmax)
     bj(j) = b
    endfragment

F2: local (i = 1,imax; j= 1,jmax)
     t1(i, j) = a(i, j) + bj(i, j)
    endfragment

F3: replicate (j = 1,jmax)
     cj(j) = c
    endfragment
```

```
F4: local (j = 1,jmax)
     linear (i = 1,imax)
      y(i)[j] = t1(i, j) + (x(i-1)[j] + cj(i, j))
      x(i)[j] = y(i)[j] * d(i, j)
    endfragment
```

Notice the partial ordering among the fragments above. F4 depends on F3 and F2. F2 depends on F1.

Consider a slight modification to the above example.

```
y(i) = a(i, j) + b(i)
x(i) = y(i) * d(i, j)
```

Here all the operations are within the same subspace $\{i, j\}$ and all are local with respect to both `i` and `j`. The two statements will form distinct innermost fragments since although they have the same subspace and expansion category, they are both local and they define distinct objects.

Multi-dimensional objects may be computed within a single fragment as in **`F4`** above, but they may also generate a hierarchy of fragments if they have complex bodies. For a complex fragment, F, the body of F may contain multiple fragments, each is F_i for some i. Each fragment, F_i, is either an innermost fragment or another complex fragment. The fragments F_i are related by a partial ordering which indicates ordering constraints imposed on their execution, just as the fragments **`F1-F4`** above are related by a partial ordering.

The reorganization performed by natural expansion category determination exposes various forms of parallelism.

- Complex and innermost fragments with expansion category local for some axis can run in parallel across that axis
- Complex and innermost fragments with the expansion category scan for some axis can be performed via parallel-prefix operations.
- Fragments consisting of expansions to operational space can be performed via a parallel fan-out.
- For complex fragments, the partial ordering of the fragments composing its body expose task parallelism. Two fragments with no ordering between them can run concurrently even if they are based on the same index. Finding each distinct cycle based on each axis allows independent cycles based on the same index to run concurrently. If each of the two cycles has N operations and the axis has extent E the first loop completes in $E * N$ operations. The second, executing concurrently one processor behind the first, needs only an additional N time for its last iteration (total $E * N + N$), whereas the original combined loop requires $E * 2N$ operations.

7 Output of the Subspace Phase

This section addresses how this internal subspace form is used as the input for the scheduler and the data partitioner. The data partitioner determines the layout of the data across the processors. If the target system is data parallel, the layout of the data determines which processors perform the operations. Otherwise, a separate scheduler determines the distribution of the operations across the processors.

7.1 Scheduler

If the target system includes a compile-time scheduler, it is the fragments, in the natural subspaces and natural expansion categories, that become the basic units of scheduling. The

scheduler will determine the appropriate distribution strategy for each loop of the fragment, presumably taking the natural expansion categories of the loops into account.

There are several advantages to scheduling after subspace analysis. The conversion to natural subspaces gives the scheduler more flexibility by eliminating unnecessary serialization due to subspaces that were too small.

User iterations may be broken into distinct fragments. This has two advantages. First each fragment can run in its natural expansion category as opposed to the most conservative category as would be required for the user iteration. In addition, the concurrency among distinct fragments can be exploited within the constrains imposed by the required partial ordering.

The scheduler also has the option of combining fragments to increase the grain size to match target machine characteristics.

7.2 Data Partitioner

The data partitioner is given the objects (named and unnamed) in their natural spaces as the objects to be partitioned. The partitioner will determine the appropriate layout for each object, presumably taking the expansion categories of the object into account.

There are several advantages to partitioning the objects that result from subspace analysis as opposed to user specified objects.

- The objects are in their natural subspace.

  Compare a user declared 1-dimensional object with its counterpart in its 2-dimensional natural subspace. If the partitioner chooses to serialize the added dimension of the 2-dimensional object, the options for partitioning the two objects are identical. But the partitioner has the added option of not totally serializing the added dimension. Since the options available for partitioning an expanded object are a superset of those for partitioning its unexpanded counterpart, the partitioner is given more flexibility.

- Each assignment to a named object is partitioned. This not only makes more options available, it also means that for any given object, there are fewer other objects that interact with it. This means fewer constraints. Consider source object `x` that interacts with `a`, `b`, `c`, `d`, `e`, `f`, and `g`. When three distinct assignments to `x` are isolated, `x1` interacts with `a`, `b`, `c` and `x2`. `x2` interacts with `d`, `e` and `x1`. `x3` interacts with `f` and `g`. Given the isolation of the three objects, the partitioner can determine that `x1`, `x2`, and `x3` are partitioned identically, but now, in addition, it has other options that might be more effective.

  This is similar to scalar renaming[5] and static single assignment[6] but applied to arrays.

- Since the data partitioner will be given an object in its natural subspace and operational expansions across distinct axes as distinct objects to be partitioned, if redistribution is required, the partitioner has the explicit choice of redistributing the original object or an expansion.

- Scalars and conditionals in the source are converted to their natural subspace and their natural expansion category and are then treated as first class objects by the partitioner. Scalars are not automatically replicated to every processor at every modification. Nor are conditional automatically replicated to every processor.

- The expansion categories for each axis for each object are available to distinguish the costs of various partitions.

After data partitioning and scheduling, the operands of the operations within a fragment may not all be aligned. Communication may be required to align them but, of course, this communication will include no reshaping, only alignment of objects within the same subspace.

7.3 Serialization

As the data partitioner takes the actual machine size into account, many of the dimensions will be totally serialized or blocked (partially serialized).

Consider an object with an axis that is expanded by subspace analysis and then serialized by the partitioner. Although the goal of the expansion was to enable parallelism, no parallelism was enabled. Since the cost of the expansion in storage is worthwhile only when the parallelism is achieved, we reverse the expansion for serialized axes.

If the axis is blocked (partially serialized) the extent of the parallelism is limited by the blocking. We limit the expansion to correspond exactly to the blocking.

8 Subspace Optimizations

We have introduced the subspace abstraction and have shown the determination of natural subspaces and natural expansion categories for each named object in the program and have discussed the form of the output to subsequent phases. In addition to these analyses, there is a class of optimizing transformations within the subspace abstraction. Several such optimizations are presented below.

For an expression tree, given the subspace and expansion categories of the leaves, we can directly determine the subspace and expansion categories of the intermediates, and we can determine the necessary expansions to operational space. For some expressions, the same result can be computed in several different ways. Expressions can be reordered, for example, via commutative and associative laws, via factoring and the distributive law and via DeMorgan's law. These reorderings can alter the subspaces and expansion categories of the intermediates and the required expansions to operational space. We present below three different optimizations. Associated with each optimization is a cost metric on which to base an optimal expression ordering.

8.1 Subspace Minimization

Some of the legal reordering will reduce the subspace of the intermediates. For example, consider the following expression with the obvious subspaces.

```
s16:  a(i) + b(i,j,k) + c(i)
```

If the expression is evaluated as `(a(i) + b(i,j,k)) + c(i)` both `+` operations will be in subspace $\{i, j, k\}$ but by reordering the expression as `(a(i) + c(i)) + b(i,j,k)`, the first `+` is now in subspace $\{i\}$. This reordering results in fewer operations for that `+` by a factor of $jmax * kmax$ and also less communication since it requires the expansion of a single value instead of two values across $\{j, k\}$ to the operational subspace of $\{i, j, k\}$. This optimization is similar in flavor to transformations performed in APL compilers to limit the computations required in the presence of subspace changes [8]. This optimization has the same functionality as loop invariant code motion combined with some transformations. But it is more direct for two reasons. First, since the operands are in their natural subspaces, no analysis is necessary to determine if the expression is invariant within a particular loop. Second, since we do not maintain the loops while we do a series of loop transformations, we need no analysis to determine if a loop transformation is legal. We generate the appropriate loops after the transformation is complete (See Section 7).

8.2 Cycle Minimization

An operation is within a cycle if one of its children is within that cycle. As with subspace minimization, reordering of the expression can improve performance, in this case by reducing the number of operations within the cycle. Consider the statement

```
s17:   a(j) = a(j-1) + b(j) + c(j)
```

where the two references to `a` are part of a cycle but neither `b` nor `c` are part of that particular cycle. If this is computed as `a(j) = (a(j-1) + b(j)) + c(j)` then both `+` operations are within the cycle. However, if we reorder it as `a(j) = a(j-1) + (b(j) + c(j))` then the sum of `b` and `c` is computable outside the cycle. Notice that the subspaces of both the intermediates are identical in both cases. We have simply made the expansion category of one of the intermediates more efficient.

8.3 Minimizing Expansions to Operational Subspace

We have shown above (See Section 4) that redundant expansions to operational subspace can be eliminated. However, we can go one step further and order the expressions to minimize the total cost of expansions to operational subspace. Consider the following two statements.

```
s18: ... = a(i) + b(j) + c(k)

s19: ... = a(i) * y(i, j)
```

Looking at `S18` alone, we have no reason to choose any one of the three possible expression orderings over any other, however, `S19` requires the expansion of `a` across `j`. This means that `a` expanded across `j` is available for free in statement `S18`. Therefore, in `S18`, adding `a(i)` and `b(j)` in $\{i, j\}$ space will be less expensive than the other two possibilities. We not only eliminate expansions that turn out to be redundant, we determine an ordering for the expressions that minimizes the cost of expansions given that we will be performing each expansion only once.

9 Relaxing Language Restrictions

In our discussion so far, we have only addressed the tiny language that includes `do` loops and assignment statements because the ideas are best illustrated within this framework. This section addresses the incorporation of additional language features, specifically I/O and predicates, into the model.

I/O statements are easily incorporated into the natural subspace computations. Input statements act as definitions of objects and output statements act as references. The subspace model uncovers potential for parallel I/O. If an input statement reaches a use that is parallel along some axis, that the input statement may be parallelized along that axis. If an output statement outputs an object that is defined in parallel along some axis, that the output statement may be parallelized along that axis.

Now we consider incorporating `if` statements with predicates.

A predicate is simply an expression and its subspace is determined as we determine the subspace for any expression. For example, in the code

```
if (a(i) .gt. b(j)) then
```

the predicate is in subspace $\{i, j\}$ if `a(i)` is in subspace $\{i\}$ and `b(j)` is in subspace $\{j\}$. It might seem that since a predicate has no explicit LHS, the propagation of subspaces would stop at the conditional expression itself. However, we treat predicates simply as additional operands of the statements they control. We will allow them to be combined with other operands producing either an arithmetic result or the reserved value, NIX. For example, in the code

```
if (a(i) .gt. b(j)) then
 x(i, j, k) = c(i, j, k) + d(i, j)
```

we have the option of incorporation the predicate, whose natural subspace is $\{i, j\}$, into the expression either within subspace $\{i, j\}$ or within subspace $\{i, j, k\}$. In the first case, `d(i, j)` and the expression `(a(i) .gt. b(j))` are combined within $\{i, j\}$. The result is a single object in $\{i, j\}$ where each element has either the corresponding value of `d` or the special value NIX. Only that single result is expanded across $\{j\}$. In the second case, both `d(i, j)` and the expression `(a(i) .gt. b(j))` are expanded across `k`. The addition and the assignment are performed locally within subspace $\{i, j, k\}$ exactly where the predicate is true. When incorporating the predicate into an expression, value NIX is a zero for any subsequent operation, that is, if any of the operands of the operation are NIX, the result is NIX. This view of predicates facilitates their use as masks in Fortran90 library operations and the use of masks in **`where`** statements.

Predicates participate in determining the subspace of the result of the operation in the usual way. Predicates also participate in expansion category determination. For example, `s20` below creates a cycle not unlike that created by `s21`.

```
s20:                              s21:
   if a(i)... then                   b(i) = a(i) + ...
      ...
      a(i+1) = ...                   a(i+1) = b(i) + ...
```

Since predicates can be combined with any operation, they can participate in reordering for subspace minimization and cycle minimization. When predicates are involved, some expression orders can generate speculative execution, where subexpressions are computed before it is known that the results will be used. This speculative aspect can be controlled by controlling where predicates are incorporated in the expression.

Predicates are also optimized by elimination of redundant expansions and, in fact, their expansions are more likely than operands of arithmetic operations to be found to be redundant since a predicate may control many statements.

Notice that with this treatment, predicates are expanded only where they are needed and no further.

We must also consider the impact of blocking and serialization by the partitioner on conditionals. As with other code, expansion dimensions are limited to one instance per processor. But with (partially or fully) serialized conditionals, we also convert from the masking style of conditional execution back to use of control flow within a given processor.

10 Conclusions

We have presented a model based on subspaces. The model subsumes specific shape related optimizations (such as invariant code motion and privatization) and shape related compilation strategies (such as owner computes and scalars are replicated). The model determines the natural subspace and natural expansion category of each named object and optimizes the subspace and expansion category of intermediates. These optimizations reduce communication, reduce the amount of computation and remove serialization restrictions. The result of these transformations determines the input to the data partitioning and code scheduling phases, that is, it determines the objects to be partitioned and the code to be scheduled. This modified input provides the scheduler and the partitioner with additional flexibility.

11 Acknowledgments

We would like to thank Bill Weihl, Joan Lukas, Fred Chong, Don Yeung, John Kubiatowicz and Carl Offner for fruitful discussions and helpful comments on earlier versions of the paper.

References

[1] S. G. Abraham, and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. In *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318-328, July 1991.

[2] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE.

[3] Jennifer Anderson and Monica Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[4] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shuang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, January 1993. Association for Computing Machinery.

[5] Ron Cytron and Jeanne Ferrante. What's in a name? -or- The value of renaming for parallelism detection and storage allocation. In *Proceedings of the International Conference on Parallel processing*, Penn State Univ. University Park, PA, August, 1987. IEEE.

[6] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and F. Kenneth Zadeck. An efficiently method of computing static single assignment form. In *ACM Sixteenth Annual Symposium on Principles of Programming Languages*, Austin, Texas, January 1989. Association for Computing Machinery.

[7] R. Eigenmann, J. Hoefinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Proceedings of the 4th workshop on Programming Languages and Compilers for Parallel Computing.* Pitman/MIT Press, AUG 1991.

[8] L. Guibas and D. Wyatt. Compilation and delayed evaluation in APL. In *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages.* Association for Computing Machinery, January 1978.

[9] HPF language specification, version 1.0. Technical Report CRPC-TR 92225, Rice University, Houston, Texas, January 1993.

[10] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[11] Kathleen Knobe and Venkataraman Natarajan. Automatic data allocation to minimize data motion on SIMD machines. *Journal of Supercomputing*, 1993.

[12] Kathleen Knobe. The subspace model: a target independent theory of shapes for parallel systems. PhD dissertation. Massachusetts Institute of Technology. (In preparation.)

[13] Jingke Li and Marina Chen. Index domain alignment: Minimizing costs of cross-referencing between distributed arrays. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, Maryland, Oct 1990. University of Maryland.

[14] Dror Maydan, Saman Amarasinghe and Monica Lam. Array Data-Flow Analysis and its Use in Array Privatization. In *ACM Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, Jan 1993.

[15] Carl Offner. A data structure for managing parallel operations. In *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume II: Software Technology*, January 1994.

[16] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[17] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. *ACM SIGPLAN Notices*, 28(1), January 1993. Proceedings of the Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors.

Efficient Distribution Analysis Via Graph Contraction

Tom Sheffler and Rob Schreiber, RIACS
Bill Pugh, University of Maryland
John Gilbert, Xerox PARC
Siddhartha Chatterjee, University of North Carolina

Array parallel languages, such as HPF, promise high performance on distributed memory parallel computers, but only if careful attention is paid to the manner in which arrays are distributed over the available processors. Currently, array distribution analysis must be performed by a programmer, who then annotates a program with distribution directives. This difficult task is further complicated because the optimal distribution for a program is often dependent on the architecture on which the program is to run. In the interest of simplifying the task of the programmer and enhancing the portability of array parallel programs, distribution analysis should be handled by a compiler.

Unfortunately, distribution analysis is a difficult combinatorial optimization problem. Heuristic algorithms can be effective for small programs. However, for very large programs or very detailed analyses (employing inter-procedural analysis, for example) these algorithms may become less effective or unacceptably slow.

In this paper, we show how to reduce the size of an instance of the distribution problem. We first cast the distribution optimization problem as a graph labeling problem and then show how parts of the graph may be eliminated through graph contraction operations. In addition to reducing the size of the distribution problem, the contraction operations serve to identify program regions that should be performed under the same distribution.

We examine different strategies for applying the contraction operations and evaluate their relative merit. Initial experiments conducted with example programs show that these contraction operations are effective in practice. However, we believe that stronger contraction operations could further reduce problem sizes to the point where they could be solved exactly.

Efficient Distribution Analysis via Graph Contraction

Thomas J. Sheffler [*] Robert Schreiber [*] William Pugh [†] John R. Gilbert [‡]
Siddhartha Chatterjee [§]

Abstract

Alignment and distribution of data by an optimizing compiler is a dream of both manufacturers and users of parallel computers. The distribution problem has been formulated as an NP-complete graph optimization problem. The graphs arising in applications are large, and the optimization problem does not lend itself to traditional heuristic optimization techniques. In this paper, we improve some earlier results on methods that use graph contraction to reduce the size of a distribution problem. We report on an experiment using seven example programs that show these contraction operations to be effective in practice; we obtain from 60 to 99 percent reductions in problem size, the larger number being more typical, without loss of solution quality.

1 Introduction

Programmers expect that array parallel languages such as High-Performance Fortran (HPF) will provide high performance on distributed memory parallel computers, if they pay careful attention to the distribution of arrays to the available processors. Currently, array distribution must be performed by a programmer, who then annotates a program with distribution directives. This difficult task is further complicated by the fact that the optimal distribution for a program is dependent on the target machine. In the interest of simplifying the task of the programmer and enhancing the portability of array parallel programs, distribution should be handled by the compiler.

Unfortunately, distribution is a difficult combinatorial optimization problem [1]. Heuristic algorithms can be effective for small programs. However, for very large programs or very detailed analyses (employing inter-procedural analysis, for example) these algorithms may become less effective or unacceptably slow.

In this paper, we show how to reduce the size of a distribution problem. We recall the formulation [1, 2] of the distribution problem as a graph labeling problem, then show how parts of the graph may be eliminated through graph contraction operations. The contraction operations are based on identifying program regions (the nodes of a subgraph) that may be performed under the same distribution. Once identified, we collapse these regions into a single node that captures all of the information present in the original problem. Our contraction operations are lossless: they do not diminish the quality of solutions that may be found.

[*] Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000. (sheffler@riacs.edu, schreiber@riacs.edu). The work of this author was supported by the NAS Systems Division via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA).

[†] Department of Computer Science, University of Maryland, College Park MD 20742. pugh@cs.umd.edu

[‡] Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304-1314. gilbert@parc.xerox.com.

[§] Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. sc@cs.unc.edu

We examine different strategies for applying the contraction operations and evaluate their relative merit. Initial experiments conducted with example programs show that these contraction operations are effective in practice. It is possible, moreover, that stronger contraction operations could further reduce problem sizes to the point where they could be solved exactly.

2 The graph model

A data-parallel program may be modeled as a directed graph (V, E). Each node $v \in V$ of the graph corresponds to an array operation in the program. An operation consumes one or more arrays, and produces one or more arrays as a result. A directed edge $(v, w) \in E$ connects a definition of an array object in array operation v to a use by operation w.

In our representation, there are also three special node types. A *fanout* node produces copies of an array with many uses; a *branch* node produces copies of an array with mutually exclusive uses, and a *merge* node combines mutually exclusive definitions of an array value (usually due to branches to a particular point in a program).

A *weight* labels each edge; it is (an estimate of) the number of elements in the array represented by the edge multiplied by an estimated trip-count of the edge in an execution of the program. In this way, the weight incorporates information about control flow.

Alignment is specified for the head and tail of each edge, giving the alignment of an array at its definition and use. Distribution must also be given for each node, specifying the distribution that is applied to each of the arrays involved in the node computation. The graph, along with these labels, is called an Alignment Distribution Graph (ADG) [3].

We allow the same mappings as HPF: an array is aligned to a template, which is distributed over the available processors. A template is an abstract array used as a target in alignment directives. An alignment is specified by four separate components. These are *axis*, *stride*, *offset*, and *replication*. *Axis* alignment determines the correspondence of array axes to template axes. *Stride* alignment specifies the factor by which the array is stretched across the template. *Offset* is a vector specifying the distance of an array from the origin of the template, and *replication* specifies certain axes of the template over which an array might be copied.

An example of alignment and distribution as specified in an HPF program follows. The first two directives declare the template and describe the alignment of the array. The third and fourth lines together describe the distribution. The `PROCESSORS` directive describes the allocation of arrays to the axes of the template, while the `DISTRIBUTE` directive specifies how the template is divided over the processors. In this case, the `BLOCK` directive says that the first dimension of the template is distributed in blocks of 25 over 4 processors, and the CYCLIC `directive` specifies that the second dimension is distributed in blocks of 10 over 8 processors. Since the extent of the template is 200 in the second dimension, the blocks will wrap around the processors.

```
real :: A(100,100)

!HPF$ TEMPLATE T(100,200)
!HPF$ ALIGN A(i,j) WITH T(i,j+100)
!HPF$ PROCESSORS P(4, 8)
!HPF$ DISTRIBUTE T(BLOCK, CYCLIC(10)) ONTO P
```

The alignment of an array object may change between its definition and use (as represented by the tail and head of an edge in the graph model). A change in alignment effectively changes the data mapping of an array and results in realignment communication. The type of communication needed to implement the realignment is determined by the component of an alignment that changes. Axis or stride realignment requires all-to-all personalized communication (AAPC), offset realignment requires shift communication, and replication realignment requires a spread (broadcast) communication operation. In general, redistribution requires AAPC.

In our model, each node in the graph is assigned a distribution. If an edge connects two nodes with different distributions, then the array carried by the edge is redistributed between its definition at the tail of the edge and its use at the head.

We choose to perform distribution analysis after alignment analysis. In our system, we first optimize axis and stride alignment, then replication alignment, followed by shift alignment. The order of the optimizations is motivated by the relative costs of the communication required by these types of realignment. It might be possible to find a better alignment if distribution information were known, but distribution analysis is difficult without some model of realignment communication costs. Consider the following example code fragment.

```
integer, parameter :: N = 1000
real a(N), left(N-2), right(N-2), cl, cr
t1 = cl * a(1:N-2)
t2 = cr * a(3:N)
a(2:N-1) = a(2:N-1) + t1 + t2
```

In our system, we would perform alignment first. The axis and stride alignments chosen here cause no realignment, but there is offset realignment in this typical finite-difference stencil computation. Because of the necessary offset realignment, which comes to light in the alignment optimization phase, we would prefer to give the arrays a block rather that a cyclic distribution, since this reduces data traffic when shift communication if performed. It would be difficult to determine this fact about distribution without having established alignment information first.

2.1 Modeling redistribution cost

The Alignment-Distribution Graph (ADG) $G = (V, E)$ may be used to model the effects of distribution decisions. Our system first finds a set D of candidate distributions. Node costs are recorded in a matrix, C, and edge costs are recorded in a matrix, W. Entry $C(d, v)$ estimates the time required to perform operation $v \in V$ under distribution $d \in D$. Realignment costs are also incorporated in this model by adding an estimate of the cost of performing the realignment, if any, on directed edge (u, v), for each distribution d, into the cost entry $C(d, u)$. Each edge, $(u, v) \in E$, has an associated weight, $W(u, v)$, which is an estimate of the time required to redistribute the array value communicated along the edge. Even though this single weight is a simplistic measure of redistribution (since it is insensitive to the actual starting and ending layouts) experiments have shown that this discrete metric accurately reflects the cost of performing a redistribution step [7].

We seek to give each ADG node a distribution in D, *i.e.*, we seek a mapping $m : V \rightarrow D$. For a particular distribution map, the cost of performing the computation of the graph is the sum over the nodes of the cost of performing their computation in the given distribution, plus the sum of the weights of all edges

whose endpoints have different distributions:

$$\text{cost}(m) = \sum_{v \in V} C(m(v), v) + \sum_{(u,v) \in E,\ m(u) \neq m(v)} W(u, v).$$

The goal of distribution analysis is to map each node to a distribution so that this cost is minimized.

The model's node cost component is trivially minimized by mapping each node to its distribution of smallest node cost; this can result in many edges carrying redistribution communication. Edge costs may be avoided entirely by mapping every node to some one distribution, the best of these being the distribution that minimizes the sum of the node costs. The optimal solution typically lies at neither of these extremes. The centrifugal tendencies toward reducing node costs (by labeling nodes independently) and eliminating edge costs (by labeling nodes identically) are what make the problem difficult. Bixby, *et al.* [1] in fact show that this formulation of the distribution problem is NP-complete.

2.2 The set of distributions

A distribution $d \in D$ specifies both the deployment of processors to the axes of the template and the blocking factor with which each axis is distributed to the processors (in a cyclic fashion). Our analysis requires a set D of candidate distributions.[1] The set may be specified by a programmer, or may be generated by a compiler as it analyzes the program. We adopt the latter approach.

The generation of a set of distributions requires care. The achieved cost is in general reduced by allowing a larger set of candidate distributions. But the running time of our optimization algorithms is sensitive to the size of D. We have previously shown how to select candidate distributions and how to limit their number [2].

2.3 Static and dynamic mappings

We introduce two terms to describe a distribution map for a subset of ADG nodes S. Let m be a given distribution map. Then S is static under m if m maps each element of S to the same distribution; S is *dynamic* under m otherwise. We say that the map m is static if V is static under m. Since all nodes in a static subset have the same distribution, no edges in a static subset carry redistribution costs and the cost of a static distribution is completely determined by the node costs.

The way in which we attempt to reduce the size of the graph is to identify *optimally static* (O.S.) subgraphs of the distribution graph.

Definition 1 (Optimally Static) *A subset $S \subseteq V$, is* optimally static *if for any map $m : V \to D$ there exists a map m' such that m' and m take identical values on $V - S$, S is static under m' and cost$(m') \leq$ cost(m).*

(Note the difference between this and the following definition: S is *weakly optimally static* if there is a minimum cost distribution map m under which S is static. The strong definition allows one to identify a family of disjoint O.S. subsets and collapse them simultaneously, the latter does not.)

Our overall plan for finding a distribution map m is this. We first determine a collection of subsets that we require to be static under m. This partially determines m. Then, for these subsets, we can ignore internal edges and aggregate them into a single "super" node whose node cost is the sum of the node costs

[1] This approach differentiates our strategy from others. For example, Wholey employed a search strategy to determine the optimal distribution for a program [8], and Gupta used heuristic methods to determine distribution parameters [6].

of its elements, thus reducing the size of the graph. Clearly, any O.S. subset should be so contracted, as this does not increase the cost of the best distribution that can be found. The remainder of the paper will be concerned with finding O.S. subsets.

Graph contraction based on O.S. subsets requires first finding a candidate subset and then testing whether it is O.S. The next section develops a theory of O.S. subsets. A later section discusses heuristic strategies for finding candidate subsets.

3 Node amalgamation for the distribution problem.

An understanding of properties of the distribution graph allows us to develop theorems that describe how subsets of nodes can be collapsed or amalgamated into super nodes, without changing the problem in an essential way. In this manner, we will reduce the size of the ADG as a first step in distribution analysis.

3.1 Definitions

Each cost matrix entry, $C(d, v)$, gives an estimate of the time required to perform the computation of node v under distribution d. It is convenient to speak of the cost table of a node, which is simply the column of entries pertaining to the node, denoted C_v. We extend this term to sets of nodes, S, where the cost table of a subset is simply the vector sum of the columns of the nodes in S, denoted C_S. Thus, $C_S(d)$ is the cost of performing all of the operations of S in distribution d.

Recall that a set of nodes S is O.S. if some optimal program distribution assigns the same distribution to all nodes in S. We may also speak of a static cost of a set of nodes, which is simply the cost of the set of nodes if they were to be placed at the same distribution. Because the cost table C_S gives the cost of the subset for each candidate distribution, we may obviously select the smallest value. The static cost of a set of nodes S is

$$\text{static}(S) = \min\{C_S(d) \mid d \in D\}.$$

We will also occasionally be interested in the worst static cost that a set could incur:

$$\text{worst}(S) = \max\{C_S(d) \mid d \in D\}.$$

The difference between the maximum and minimum cost of a single node is called the *range* of the node.

$$\text{range}(v) = \max_D C_v(d) - \min_D C_v(d).$$

Let a subset $S \subseteq V$ be given. We shall use a simple lower bound on the cost of a distribution map under which S is dynamic. Such a distribution map could potentially assign every element of S to its minimum cost distribution; or it could avoid redistribution costs on edges leaving S at the expense of some redistribution cost internally (assuming that S is connected). In either case, a dynamic solution divides S into at least two partitions with different distributions, and every edge crossing the partition carries redistribution communication. This redistribution cost is at least as great as the weighted minimum-cut (min-cut) of the undirected subgraph induced by S. Thus, for a distribution map m such that S is dynamic under m,

$$\text{dynamic}(S) \equiv \sum_{v \in S} \min C_v + \text{mincut}(S)$$

is a lower bound on cost(m).

Many of our proofs require consideration of the edges crossing from one set S to another set T. Define the function weight(S, T) as follows:

$$\text{weight}(S, T) = \sum_{v \in S, w \in T} W(v, w) + W(w, v).$$

Thus, weight$(S, \bar{S})$ is the sum of the weights of all edges entering or leaving S. We will also need a function that determines the maximum weight with which a set is attached to any particular neighbor.

$$\text{maxweight}(S) = \max_{v \notin S} \text{weight}(S, \{v\}).$$

3.2 Optimally static subsets

We present a number of tests that may be used to verify that a subset of nodes is O.S. Each of the lemmas below gives an explicit construction showing, for a class of subsets S, how a map with dynamic S can be modified on S to make S static and not increase the cost. A following section discusses the implementation of the tests and the expected running time of each.

Lemma 1 (Accretion) *[2] Let S be O.S. and assume $v \notin S$. If*

$$\textit{weight}(\{v\}, \bar{S}) + \textit{range}(v) \leq \textit{weight}(\{v\}, S)$$

then $S \cup \{v\}$ is O.S.

Proof: Any map may, by assumption, be modified on S to make S static without increasing its cost. Now consider a map in which S is static with distribution d and v has a different distribution d'. Changing the distribution of node v to d reduces the cost of the mapping by $w(v, S)$ and raises it by at most $w(v, \bar{S}) + \text{range}(v)$. By the hypotheses, this change also does not increase the cost. Hence $S \cup \{v\}$ is O.S. □

Corollary 1 (Series) *Given a node y with only two distinct neighbors, x and z, the set $S = \{x, y\}$ is O.S. if $W(y, z) + \textit{range}(y) \leq W(x, y)$.*

Proof: Since any singleton node is an O.S. subset, the corollary follows immediately from Lemma 1. □

This trivial corollary of Lemma 1 turns out to be very useful in practice: it identifies pairs of nodes that should be merged. In particular, unary operations representing SPREAD and REDUCE functions often have small ranges and have input and output edges of very different weights. Elementwise unary operations may have zero range with equal weights on their two incident edges.

Lemma 2 (Min-cut) *[2] A set S is O.S. if $\textit{static}(S) + \textit{weight}(S, \bar{S}) \leq \textit{dynamic}(S)$.*

Proof: Assume that S is dynamic under a given distribution map. If the inequality holds, then the cost of this map is not increased by assigning S to its best static distribution, incurring the static node cost as well as potential redistribution on all edges leaving S. □

The strategy of the previous lemma was to remap all of S to its preferred single location. As an alternative, we consider remapping all of S to the distribution of one of its neighbors, so as to make the union of S and that neighbor static. Define adj(S) to be those nodes outside of S with one or more neighbors in S.

Lemma 3 (Adjacent Vertex) *Let $S \subseteq V$. If*

$$(\textit{worst}(S) - \sum_{v \in S} \min C_v) + \textit{weight}(S, \bar{S}) - \textit{maxweight}(S) \leq \textit{mincut}(S), \tag{1}$$

then S is O.S.

Proof: Let S be dynamic under some map m, and let v be as in the hypotheses. Remap all nodes in S to the distribution of node v. The node costs can increase to the worst static cost of S, thus increasing by no more than the first term of the inequality 1; the edges from S except those touching v may now incur redistribution costs. By the hypothesis, these added costs are at least recompensed by the absence of redistribution along the edges internal to S. □

Note that although the construction in the proof guarantees that $S \cup \{v\}$ is static after the relabeling, we cannot conclude that $S \cup \{v\}$ is O.S., since we claimed a gain of mincut(S) after relabeling a map for which S is dynamic. The following reformulation allows us to conclude that $S \cup \{v\}$ is O.S.

Lemma 4 (Border) *Let $S \subseteq V$. If*

$$(\textit{worst}(S) - \sum_{v \in S} \min C_v) + \textit{weight}(S, \bar{S}) - \textit{maxweight}(S) \leq \textit{mincut}(S \cup \{v\}), \tag{2}$$

then $S \cup \{v\}$ is O.S.

Nodes that do elementwise computation often do not prefer any distribution to any other, as long as all the distributions in D balance the load (number of elements per processor).

Definition 2 (Apathy) *A set of nodes, S, is* apathetic *if range$(v) = 0$, for all $v \in S$.*

For a set S of apathetic nodes, worst$(S) = \sum_{v \in S} \min C_v$, so the first term of inequalities 1 and 2 vanish.

When S is an apathetic singleton set. Lemma 4 holds when one of the edges touching the node outweighs the sum of all others. In this simplified form, this lemma is useful for identifying single nodes that should be merged with a heavily connected neighbor into an O.S. subset.

4 Locating subsets

The lemmas developed in the preceding section verify that a subset of nodes is O.S., but do not reveal how to find candidate subsets. It is clearly impractical to consider all possible subsets of V, so we develop heuristics to help locate subsets with the potential to be O.S.

Lemmas 2, 3 and 4 compare static costs to dynamic costs. Static distributions must tolerate redistribution communication on edges leaving the subsets, while the dynamic distributions only incur the min-cut cost on edges internal to the set. In order for a subset to pass any of these tests, it must be highly connected internally (leading to a large min-cut value), with low-weight connections to nodes outside of the subset. Using this observation, we construct the connected components of the subgraph obtained by deleting from E all edges whose wieght is less than or equal to some specified threshold, t. The mincut of any such componenet is therefore not less than t, while the weight of each external edge is less than t.

We examine a set of thresholds, T, which is generated by histogramming the edge weights of the graph and dividing the histogram into buckets. The minimum value in each bucket becomes a threshold value in

the set T. To use this algorithm, we work through the thresholds in T from heaviest to lightest. We apply the O.S. tests to each connected component at the current threshold.

This heuristic is effective because of the way in which the edge weights of the ADG are calculated. Recall that the ADG incorporates the effects of control flow into its weight calculation by multiplying the weight of an edge by its estimated trip count. In the ADG, nodes corresponding to operations within loops are connected by high weight edges, and values are communicated into and out of loops by low weight edges (because they are traversed only once). The strategy above tends to find connected components encompassing the operations inside the bodies of loops. The buckets tend to correspond to different levels in loop nests.

The complexity of this subset finding algorithm is proportional to the number of edges, $|E|$, and the number of thresholds, $|T|$. The histogramming phase of the algorithm can be performed in time proportional to $|E|$, and connected components can be found in time $t = O(|E|)$ by using depth-first search. The enumeration of all subsets using this technique can be performed in $t = O(|T|\,|E|)$ time.

4.1 A slight modification

The procedure above locates subsets based only on edge weights, but it also beneficial to incorporate cost table information into the heuristic. If we look at the components of the static and dynamic costs due to the cost table entires, we note that we would like to minimize the following difference:

$$\Delta(S) = \min_{d \in D} \sum_{v \in S} C(d, v) - \sum_{v \in S} \min_{d \in D} C(d, v).$$

Elsewhere, we have called this quantity the "dissension" of a set of nodes[2]. A set has a dissension of 0 if all nodes agree on the best distributon.

We initially tried modifying the subset selection procedure to find subsets with a dissention of 0, but this strict criterion did not work well in practice. We also tried heuristics for find subsets having low dissension; these approaches are still under investigation.

5 Implementing the O.S. tests and the contraction operation

This section suggests data structures and algorithms for implementing the tests of the preceding section. While none of the algorithms presented is difficult to implement, a naive implementation of the tests could lead to poor running times. In particular, dense representations of the matrices C and W are inappropriate not only because of the waste of storage that would occur, but because suitable graph traversals are not supported. Sparse matrix algorithms are necessary because of the sparse structure of the ADG. Contraction operations maintain sparsity as well.

5.1 Data structures

The matrices C and W are stored as sparse matrices. An element in a matrix is a record structure storing its row, column, and value, and pointers threading it into two doubly-linked lists: a list of elements in the same row, and another list of elements in the same column. For each matrix, two vectors of pointers record the first element in each row and column. The elements of the lists are unordered.

Finding a particular matrix element in this data structure requires potentially searching through an entire row or column list. However, our algorithms do not require finding individual elements quickly, but rather

depend on a data structure that supports efficient traversals of various types. The main operations required are the following.

insert: Given a pointer to an matrix element, insert it into the appropriate row and column lists. This may be accomplished in $O(1)$ time.

delete: Given a pointer to an matrix element, delete it from both of its lists. This may be accomplished in $O(1)$ time.

neighbors: Enumerate the neighbors of a given node, v. These may be enumerated by iterating through the row and column lists of v in the weighted adjacency matrix, W.

A set of nodes S is represented in a simple linked list structure so that its nodes may be enumerated. Our algorithms also require two mark vectors, called $mark$ and $border$, each of length $|V|$. These two vectors are cleared once at the beginning of the program. Some of the O.S. tests will set and clear marks in these vectors, but none will be required to touch the entire vector.

Our algorithms use a Sparse Accumulator (SPA) to add sparse vectors [4]. A sparse vector is a sequence of `<index,value>` pairs, where the largest possible index is some value N. The SPA maintains 3 internal vectors of length N called `accum`, `flags` and `elts`. At program startup time, these vectors are allocated and cleared. The vector `accum` holds the values of all elements of a sparse vector, `elts` is used as a stack that records the indices of nonzero elements in `accum`, and `flags`, a boolean vector, is set if the corresponding element has been pushed onto `elts`. A SPA is used to compute the sum of several sparse vectors in time proportional to the number of nontrivial arithmetic operations actually performed.

5.2 Contracting nodes

The node contraction operation replaces a set of nodes, S, with a single node s in a reduced graph. The node cost vector C_S of the new node is the sum of the node costvectors of its elements, and the weight of each edge incident to S is the sum of the weights of all edges between the adjacent node and elements of S. Precisely, this is written as

$$C(d,s) = \sum_{v \in S} C(d,v), W(s,v) = \sum_{v \in S, w \notin S} W(v,w), W(v,s) = \sum_{v \in S, w \notin S} W(w,v).$$

Our technique contracts S by adding the sparse vectors that encode the edge weight and adjacency information for the nodes in S. Merging the cost table entries for the nodes requires adding the corresponding columns of C. Thus, the cost table entries for a set can be computed in $O(|S| \cdot |D|)$ time.

Merging the adjacency table entries requires merging both the row and column lists for the nodes of S. A row or column is a sparse vector, and so we may use a SPA to implement the merger. In the row merge phase, add each element of the S row lists into the SPA in unit time per element. Enumerate the elements of the SPA, creating a new matrix element for each and insert it into its row and column lists in unit time. Clear the SPA when done. Column contraction proceeds in the same manner.

Define degree(v) as the number of nodes adjacent to node v in the graph, and let $n = \sum_{v \in S}$ degree(v). The total running time of the contraction of a set of nodes is $O(|S| \cdot |D| + n)$.

5.3 Series test

The series test is easy to implement. Determine if a node is a series node by traversing its adjacency list. If it has two or fewer neighbors, record the edges and the weights. Compute the range of the node by examining its $|D|$ cost table entries. The series test can be applied all nodes in $O(|V| \cdot |D|)$ time.

We will show later that the series test is effective at reducing the size of the graph. Because it is so simple to implement, this test should always be used.

5.4 Apathetic test

We only apply this test to single nodes, even though it is defined for sets. Determine if a node is apathetic by examining each of its $|D|$ cost table entries. If the node is apathetic, then visit its neighbors, keeping a running sum, w, of the weights of all edges encountered, and recording the heaviest edge, ***maxweight***. If ***maxweight*** $\geq (w -$ ***maxweight***$)$, then merge the node with its most heavily connected neighbor. The apathetic test can be applied to all nodes in time $O(|V| \cdot |D| + |E|)$ time.

5.5 Dynamic cost

The rest of the tests require computing the dynamic cost of a set of nodes, S. Clearly, the sum of the minima of the node costs can be computed in $O(|S| \cdot |D|)$ time. The difficult part is computing the min-cut value. There are two options: use an easily obtained lower bound on the min-cut, or compute it exactly.

If S is connected, the mincut(S) is not less than than the minimum weight edge in S. There are at most $|S|^2$ edges and we may find the lightest one by examining all edges internal to S. To be precise, we first enumerate S setting $mark[v]$ for each v in S, and then look at all edges connected to vertices in S. Any whose other endpoint is marked in $mark$ is an internal edge. We must also clear $mark$ when we are done. The running time of this computation is $O(|S|^2)$.

In the second case, we compute the global min-cut of the set exactly, using an algorithm of Goldberg and Tarjan which runs in $O(|S|^4)$ time [5]. In practice, when using this option, we only invoke the min-cut procedure when the size of the set is smaller than some predefined value – because the running time of the min-cut procedure becomes unnaceptable for large sets.

In summary, an estimate of the dynamic cost of a set of nodes may be found in $O(|S| \cdot |D| + |S|^2)$, or $O(|S| \cdot |D| + |S|^4)$ time. In estimating the time of the following tests, we will simply write $dyn(S)$ as the running time for finding the dynamic cost of a set of nodes.

5.6 Best static test

The static cost of a set of nodes is computed in $O(|S| \cdot |D|)$ time by first creating the cost vector for the set of nodes and then finding the minimum value. Computing the weight of edges leaving S requires first setting $mark[v]$ for each $v \in S$ and then traversing all edges adjacent to vertices in S. This test may be implemented in $O(|S| \cdot |D| + \sum_{v \in S} \text{degree}(v) + dyn(S))$ time.

5.7 Border outside test

The worst static cost of a set of nodes is computed as easily as computing the best static cost, and the weight of all the edges leaving S can be computed as before. This test is different in that all vertices external to S must be visited in turn. When computing the weight of edges external to S, set $border[v]$ for any edge

encountered connecting S to an external node v. If $border[v]$ was not set when encountered, then add v to the set of border nodes called B. (The marks may be cleared by traversing B at the end of the Border Outside test.) For each external border node, compute the weight that connects it to S using the $mark$ vector to determine nodes in and out of S in $O(\text{degree}(v))$ time for each border node. The running time of this test is $O(|S| \cdot |D| + \sum_{v \in S} \text{degree}(v) + \sum_{v \in B} \text{degree}(v) + dyn(S))$. This test is not much more expensive than the Best Static test if B is not too big.

5.8 Border inside test

The Border Inside test is similar to the Border Outside test except that for each node v in the border we must compute worst$(S - \{v\})$ and $\min C_v$. This may be done in the following way to make the test run in the same time as the previous one. Initially, instead of computing worst(S), record the cost vector for the set, C_S. Find the border nodes as before. Now, as each border node is visited, make use of the fact that worst$(S - \{v\}) = \max(C_S - C_v)$ and compute both this value and $\min C_v$ in $O(|D|)$ time. The rest of the implementation of the test is the same as the Border Outside test and can be implemented in $O(|S| \cdot |D| + \sum_{v \in S} \text{degree}(v) + \sum_{v \in B} \text{degree}(v) + dyn(S))$ time.

6 Experiments

We now present an experimental study of the effectiveness of the contraction operations developed earlier. The process of locating subsets and verifying that they are O.S. is heuristic; such a study is therefore mandated, and we view the data below as preliminary, pending better tools and a larger base of experimental programs.

Using program analysis tools we have developed earlier [3], we constructed the distribution graphs for seven test programs and applied various combinations of the contraction operations. The contraction operations are sensitive to the adjacency structure of the graph as well as values of the cost entries. For this reason, it is important to understand how the test cases were generated. We begin by describing the example programs and how the cost values were calculated. We then discuss contraction strategies and examine the results of these strategies.

6.1 The example programs

We chose seven example programs that represent typical scientific applications. A brief description of each of the seven follows. In addition, Table 1 describes properties of the cost and adjacency tables for each of the programs. Each of the graphs is quite sparse. With the exception of `BlockLU`, each program was analyzed with a relatively small number of distributions. Because BlockLU has many different feature sizes, a large number of distributions are generated by our automatic system.

`ADI`: A two-dimensional alternating-direction implicit (ADI) algorithm. This algorithm uses cyclic reduction to solve tridiagonal systems.

`BlockLU`: A blocked algorithm for LU factorization of a dense matrix.

`Erle`: A three-dimensional alternating-direction implicit (ADI) algorithm. This differs from the one above in that it uses Gaussian elimination to solve the recurrences.

`LU`: The LU factorization on a dense matrix.

Table 1: Properties of the example program graphs. Each is quite sparse. In general, the number of distributions used in the analysis of each program is relatively small, with the exception of `BlockLU`.

Properties of the Programs			
Program	$\|V\|$	$\|E\|$	$\|D\|$
`ADI`	232	308	12
`BlockLU`	108	131	41
`Erle`	666	845	7
`LU`	21	25	12
`Shallow`	445	545	3
`Tred`	105	124	9
`TwoZone`	335	411	12

`Shallow`: A benchmark weather prediction program; finite-difference approximation of the the shallow water equations.

`Tred`: Reduction of a dense matrix to tridiagonal form using Householder transformations.

`TwoZone`: Solution of Poisson's equation in an L shaped domain by Schwartz alternating procedure, using a JOR (Jacobi Over-Relaxation Method) for the subdomain solver.

6.2 Cost table construction

In Section 2, we differentiated between three communication patterns: all-to-all personalized communication (AAPC), offset communication (shift), and reduction/replication communication. When analyzing a program, we estimate the time of an elementwise operation to be proportional to the amount of data on the most heavily loaded processor. We estimate the time of a communication operation to be proportional to the maximum amount of data sent or received by any one processor, with the constant of proportionality determined by the type of operation. The three constants are ρ (for AAPC), σ (for reduction/replication) and ν (for shift). (The names recall the now ancient and disappearing Connection Machine jargon: *router, scan, NEWS*). High-level operations in HPF give rise to one of these three types of low-level communication. Table 2 shows the correspondance between high-level and low-level communication operations.

In general, it is impossible to predict how varying the parameters, ρ, σ, and ν, will affect the contraction operations. Even the interaction between this model of communication and the cost values generated is quite complex. Realignment costs are incorporated into the node cost table, while redistribution costs affect adjacency information. Varying the parameters by the same factor changes the relationship between elementwise computation and communication. Varying the parameter ρ can affect values in both, while varying σ or ν can only affect values in the cost table. Because of these complex interactions, we ran tests of the contraction operations for a number of values of the parameters to see how the results changed.

6.3 Contraction operation strategies

The contraction operations may be applied individually, or in combinations. In the discussion of the combinations of contraction operations we will use a shorthand. The character "a" means an application

Table 2: Mapping of high level HPF operations to low-level communication types. Each of the three low-level operations is modeled as requiring time proportional to the amount of data communicated, with the constant of proportionality as shown.

Coefficients of Proportionality		
High-Level Operation	Low-Level Communication Type	Constant
Redistribute	AAPC	ρ
Stride Realign	AAPC	ρ
Axis Realign	AAPC	ρ
Offset Realign	shift	ν
Replication Realign	broadcast	σ
Subscript	AAPC	ρ
Reduction	fan-in	σ

of the Apathetic test, an "s" the Series test. The character "S" stands for the Best Static test, and "B" for the Border tests. Each of the last two tests are parameterized by the subset selection method used, "cc" or "ccc", and the method of estimating the min-cut. The "cc" subsets are connected components of edge-weight thresholded subgraphs. The "ccc" data employ a heuristic designed to improve the dissention of these subgraphs by removing additional edges before finding the connected components. The keyword "min" means the minimum weight edge in the subset was used as a bound, and an integer P means that an exact min-cut was computed for subsets of size less than P.

6.4 Results

For each of the seven programs, we generated test data assuming a 64 processor target using three sets of values for the communication parameters as shown below.

Case1	$\rho = 1$	$\sigma = 1$	$\nu = 1$
Case2	$\rho = 10$	$\sigma = 1$	$\nu = 1$
Case3	$\rho = 100$	$\sigma = 10$	$\nu = 1$

Case 1 reflects a target architecture where communication costs as much as computation. There is no such machine widely available today, but such a machine would tolerate a lot of redistribution, prefering dynamic distributions over static ones. Thus, this case should thwart many of our contraction operations.

Case 2 reflects a case where communication is only slightly expensive. Scans and shifts are still very inexpensive. Case 3 is the most realistic model of current machines. Shift communication is very cheap, but any general AAPC communication is quite expensive. We would expect both of these cases to enourage static solutions to the distribution problem, and thus expect our contraction operations to do well.

Tables 3, 4 and 5 show the results of the contraction experiments for the three different cases. The size of the original program is shown at the top of each column, and the results of applying nineteen different contraction combinations below. The combinations are divided into groups. The first group includes only the single-vertex tests ("a" and "s"). The next group shows the subset tests ("S" and "B") alone. The next groups show the "S" or "B" tests preceded by the "as" combination. It is clear from the data

Table 3: The size of the contracted graphs with the communication parameters of Case1. With ρ set low, we expect to see graphs that prefer dynamic distributions, thus it should be difficult to prove any that any subgraphs are O.S. Many of the test graphs are contracted significantly.

Contraction Results for Case 1: $\rho = 1, \sigma = 1, \nu = 1$							
method	ADI	Block	Erle	LU	Shal	Tred	TwoZ
ORIGINAL	232	108	666	21	445	105	335
s	82	41	295	5	125	32	150
a	220	84	548	16	339	77	310
as	82	41	289	5	125	32	150
asas	81	39	285	5	125	29	150
S(cc,min)	232	57	555	3	445	80	73
S(ccc,min)	232	84	552	21	441	105	319
B(cc,min)	224	84	555	13	445	80	273
B(ccc,min)	224	72	529	11	425	79	317
asS(cc,min)	82	17	253	2	22	23	2
asS(cc,25)	42	10	253	2	22	8	2
asS(cc,50)	42	10	253	2	22	8	2
asS(ccc,min)	82	29	253	4	125	29	74
asB(cc,min)	74	27	253	4	37	23	105
asB(cc,25)	74	24	253	2	37	23	74
asB(cc,50)	74	24	253	2	37	23	74
asB(ccc,min)	74	30	253	3	125	23	105
asB(ccc,25)	72	27	253	3	114	21	102
asB(ccc,50)	72	27	253	3	114	21	74
asS(cc,min)B(cc,min)	72	17	253	2	22	23	2

that the combination that nearly always achieves the best contraction of all those tried is the "asS(cc,50)" combination. The "asS(cc,min)" combination also fared well many of the times, but occasionally the true min-cut values were critical to obtaining further contraction.

The contraction programs that involve only the Apathetic and Series contraction operations ("a" and "s") are actually very effective considering just how inexpensive they are to implement. Recall that these tests examine only the neighbors of single nodes. These two contraction operations should *always* be applied first, as they are *enabling* contractions for the more powerful Best Static and Border contractions. Notice that in most of the test cases, the Best Static or Border tests alone are ineffective, while pre-contracting with the Apathetic and Series tests helps them considerably. Also notice that repeating the Apathetic and Series tests ("asas") is not worthwhile.

The overall percentage of reduction achieved by the "asS(cc,50)" combination is shown in Table 6. Initially, we did not expect to be able to contract the graphs very much for Case1 because redistribution is fairly inexpensive in this case. With low edge-weights, we did not expect to find many O.S. subgraphs. The results show that, on the contrary, the tests are effective even when ρ is small.

In most of the trials, we used the Best Static and Border tests in a mutually exclusive manner. The final case chains the two together. In almost all of the trials, we observe the the results produced by the Border

Table 4: The size of the contracted graphs with the communication parameters of Case2.

Contraction Results for Case 1: $\rho = 10, \sigma = 1, \nu = 1$							
method	ADI	Block	Erle	LU	Shal	Tred	TwoZ
ORIGINAL	232	108	666	21	445	105	335
s	68	35	283	5	125	32	144
a	220	84	548	16	339	77	310
as	68	35	277	5	125	32	144
asas	67	33	273	5	125	28	144
S(cc,min)	232	65	555	3	445	16	73
S(ccc,min)	232	84	552	21	441	105	319
B(cc,min)	232	62	555	9	445	80	153
B(ccc,min)	226	54	529	7	425	81	319
asS(cc,min)	68	20	46	2	22	8	2
asS(cc,25)	68	2	46	2	22	8	2
asS(cc,50)	68	2	21	2	22	8	2
asS(ccc,min)	68	24	241	4	125	29	70
asB(cc,min)	68	19	241	2	22	23	37
asB(cc,25)	68	12	241	2	22	23	37
asB(cc,50)	68	12	241	2	22	23	37
asB(ccc,min)	66	20	241	3	125	21	70
asB(ccc,25)	66	11	235	3	114	21	70
asB(ccc,50)	66	11	235	3	114	21	70
asS(cc,min)B(cc,min)	68	19	46	2	22	8	2

Table 5: The size of the contracted graphs with the communication parameters of Case3.

Contraction Results for Case 1: $\rho = 100, \sigma = 10, \nu = 1$							
method	ADI	Block	Erle	LU	Shal	Tred	TwoZ
ORIGINAL	232	108	666	21	445	105	335
s	68	35	283	5	125	32	144
a	200	84	548	16	339	77	310
as	68	35	277	5	125	32	144
asas	67	33	273	5	125	28	144
S(cc,min)	232	87	555	3	445	16	73
S(ccc,min)	232	90	552	21	441	105	335
B(cc,min)	232	81	555	3	445	80	73
B(ccc,min)	226	60	529	20	421	73	335
asS(cc,min)	5	24	241	5	22	8	2
asS(cc,25)	5	3	235	2	22	8	2
asS(cc,50)	5	3	235	2	22	8	2
asS(ccc,min)	68	28	241	4	125	29	144
asB(cc,min)	16	18	241	2	22	23	2
asB(ccc,min)	66	18	241	3	125	21	144
asB(ccc,25)	64	14	241	32	114	21	136
asB(ccc,50)	64	14	241	32	114	21	136
asB(cc,25)	16	3	241	2	22	8	2
asB(cc,50)	16	3	241	2	22	8	2
asS(cc,min)B(cc,min)	5	18	241	2	22	8	2

Table 6: The amount of contraction as a percentage of the total size for the combination "asS(cc,50)." This particular combination proved the most effective overall.

Percentage Contraction using asS(cc,50)							
	ADI	Block	Erle	LU	Shal	Tred	TwoZ
Case1	82%	91%	62%	90%	95%	92%	99%
Case2	71%	98%	95%	90%	95%	92%	99%
Case3	98%	97%	65%	90%	95%	92%	99%

tests are worse than those produced by the Best Static test. This is not surprising, because the Border tests involve the *worst* static value which is generally a loose bound. What is surprising is that in some cases the Border tests do quite well.

7 Conclusions

When we began formulating algorithms for solving the distribution problem, we originally felt that sophisticated optimization techniques would be needed. We now believe that contraction operations can dramatically reduce the size of a distribution problem without losing information. With effective contraction operations, problem sizes become so small that less powerful optimization strategies may suffice. Indeed, some problems become small enough that it may be possible to find optimal solutions exactly.

Some issues that remain open are these. Should one relax the requirement that the contraction operations remain lossless — contraction operations may be accepted for subgraphs that are not necessarily O.S. What is the tradeoff, if this is done, between compile time and run-time? Is it better to do a heuristic optimization of a big but exact distribution problem or an exact optimization of a small but approximate problem? We also need to reexamine our subset selection procedure. It is interesting whether, in the few cases in which the contracted graph remains large, the reason is that we haven't found the right subsets to test, or our lemmas are not powerful enought to detect that the sets we select *O.S.*, or simply that there aren't any more O.S. sets left to be found.

References

[1] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1993.

[2] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. In K. Pingal, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 76–91, Ithaca, NY, August 1994. Springer-Verlag. Also available as RIACS Technical Report 94.09.

[3] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Modeling data-parallel programs with the alignment-distribution graph. *Journal of Programming Languages*, 2:227–258, 1994. Special issue on compiling and run-time issues for distributed address space machines.

[4] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, January 1992.

[5] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[6] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1992. Available as technical reports UILU-ENG-92-2237 and CRHC-92-19.

[7] P. Hough and T. J. Sheffler. A performance analysis of collective communication on the CM-5. Excalibur project meeting note.

[8] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991. Available as Technical Report CMU-CS-91-121.

Automatic data mapping and program transformations

Jagannathan Ramanujam, Louisiana State University

This talk presents a technique for finding good distributions of arrays and suitable loop restructuring transformations so that communication is minimized in the execution of nested loops on message passing machines. For each possible distribution (by one or more dimensions), we derive the best unimodular loop transformation that results in block transfers of data. Unlike other work which focus on either data layout or on program transformations, this talk combines both array distributions and loop transformations resulting in good performance. The techniques described here are suitable for dense linear algebra codes. In addition, we present a method for deriving alignments and minimizing communication costs due to realignments.

Automatic Data Mapping and Program Transformations

J. Ramanujam* and A. Narayan
Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803
jxr@gate.ee.lsu.edu

Abstract

This paper presents a technique for finding good distributions of arrays and suitable loop restructuring transformations so that communication is minimized in the execution of nested loops on message passing machines. For each possible distribution (by one or more dimensions), we derive the best unimodular loop transformation that results in block transfers of data. Unlike other work which focus on either data layout or on program transformations, this paper combines both array distributions and loop transformations resulting in good performance. The techniques described here are suitable for dense linear algebra codes.

1 Introduction

On a distributed memory machine, local memory accesses are much faster than accesses to non-local data. Inter-processor communication—resulting accesses to non-local data—is a major determinant of the performance of a parallel machine. When a number of non-local accesses are to be made between processors, it is preferable to send fewer but larger messages rather than several smaller messages more frequently (called *message vectorization*). This is because the message setup cost is usually large. Even in shared memory machines, it is preferable to use block transfers.

Given a program segment, our aim is to determine the computation and data mapping onto processors. Parallelism can be exploited by transforming the loop nest suitably and then distributing the iterations of the transformed outermost loop onto the processors. The distribution of data onto processors may then result in communication and synchronization which counters the advantages obtained by parallelism. This paper presents an algorithm which results in the optimal performance while simultaneously considering the conflicting goals of parallelism and data locality.

While a programmer can manually write code to enhance data locality by specifying data distribution among processors, we present a technique where we can automatically derive data distribution given the program structure. We present a method by which the program is restructured such that when the outer loop iterations are mapped onto the processors, it results in the least communication. Wherever communication is unavoidable, we restructure the inner loop(s) so that data can be transferred using block transfers; such an approach is referred to as *message vectorization.*

*Supported in part by an NSF Young Investigator Award (CCR–9457768), NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09. A version of this paper appers in *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing,* 1995.

In this paper, we consider the cases where we allocate outer iterations to processors so that each outer loop iteration is done by a single processor. The data is then allocated so that there is minimum communication and all communication is done through block transfers. This paper deals with an algorithm to restructure the program to enhance data locality while still enabling parallelism. We construct the entries of a legal invertible transformation matrix so that there is a one-to-one mapping from the original iteration space to the transformed iteration space. This transformation when applied to the original loop structure will do the following:

- Allow the outermost most loop iteration to be distributed over the processors *i.e.,* an entire outermost iteration is mapped on to a single processor.
- Determine the data distribution (block or cyclic distribution of a single array dimension).
- Allow blocks transfers to be moved out of the innermost loop so that all the necessary data are transferred to the respective local memories before the execution of the innermost loop.

2 Background and Terminology

The transformation matrix is derived from the *data reference matrix* of the array references. Given a loop nest with indices $i_1, i_2, \cdots, i_n$ which is represented by a column vector $\vec{I}$, we define a *data reference matrix*, $A_{\mathcal{R}}$, for each array reference A (distinct or non-distinct) in a loop nest such that the array reference can be written in the form $A_{\mathcal{R}}\vec{I} + \vec{b}$ where $\vec{b}$ is the offset vector.

Example 1:
for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    for $k = 1$ to N_3 do
      $B[i, j - i] = B[i, j - i] + A[i, j + k]$

In the above example, the data reference matrix for the array B is

$$B_{\mathcal{R}} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{pmatrix},$$

and the data reference matrix for array A is

$$A_{\mathcal{R}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

Note that there are two data reference matrices for the array B though they are identical. For each array, we use only the distinct data reference matrices.

2.1 Effect of a Transformation

On applying a transformation T to a loop with index I, the transformed loop index becomes $\vec{I'} = T\vec{I}$ and the transformed data reference matrix becomes $A'_{\mathcal{R}} = A_{\mathcal{R}}T^{-1}$. The columns of T^{-1} determine the array subscripts of the references in the transformed loop. The key aspect of the algorithm presented in this paper is that the entries of the inverse of the transformation matrix are derived using the data reference matrices.

2.2 Related work

Li and Pingali [7] discuss the completion of partial transformations derived from the data access matrix of a loop nest; the rows of the data access matrix are subscript functions for various array accesses (excluding constant offsets). Their work assumes that all arrays are distributed by columns. In contrast, our work attempts to find the best distribution for various arrays (by rows, columns, or blocks) such that communication incurred is minimal; for each possible combination of distribution of arrays, we find the best compound loop transformation that results in least communication. Among all these possible distributions (and the associated loop restructuring), we find the one that incurs the smallest communication overhead. Several researchers have addressed the issue of automatic alignment [2, 3, 4, 5, 6, 8, 10]. None of these except [1] addresses the interaction of program transformations and data mapping.

2.3 Motivation

Consider Example 1 above which is similar to the one in [7]. There are two references to the array B (though not distinct) and one reference to the array A. Li and Pingali [7] assume that all arrays are distributed by columns and derive a transformation matrix that matches column distribution. In this case, the loop can be distributed in such a way that there is no communication incurred. Both the arrays can be distributed by rows, *i.e.,* each processor can be assigned an entire row of array A and an entire row of array B. This makes the loop run without any communication. We notice that the first row in the data reference matrix for the arrays A and B are the same *i.e.,* $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$. This allows the first dimension of both the arrays to be distributed (*i.e.,* by rows) over the processors so that there is no communication. In the next section, we derive an algorithm to construct a transformation matrix, which determines the distribution of data.

3 Algorithm

We restrict our analysis to affine array references in loop nests whose upper and lower bounds are affine. We assume that the iterations of the outermost loop are distributed among processors. To exploit data locality and reduce communication among processors, we further look at transformations that facilitate block transfers so that the data elements which are referenced are brought to local memory in large chunks; this allows to amortize the high message start-up costs over large messages. We assume that the data can be distributed along any one dimension of the array (wrapped or blocked). The results can be easily generalized where data is distributed along multiple dimensions and block transfers set up in outer iterations.

3.1 Criteria for Choosing the Entries in the Transformation Matrix

Let the array indices of the original loop be $i_1, i_2, \ldots, i_n$. Let the array indices of the transformed loop be $j_1, j_2, \ldots, j_n$. We look for transformations such that the LHS array has the outermost loop index as the only element in any one of the dimensions of the array, *e.g.* $C(*, *, \ldots, j_1, \ldots, *)$ where j_1 is in the r^{th} dimension and "$*$" indicates a term independent of j_1. The LHS array can then be distributed along dimension r. This means that the data reference matrix $C'_{\mathcal{R}}$ of the transformed array reference C, has at least one row in which the first entry is non-zero and the rest are zero, *i.e.,* there is a row r in $C'_{\mathcal{R}}T^{-1} = (\alpha, 0, 0, \ldots, 0)$. For all arrays that appear on the right hand side:

- If a row in all the data reference matrices of an array is identical to a row in the reference matrix in the LHS array, then that array can be distributed in the same way as the LHS

Step 0: If a row in the reference matrix of all the arrays are the same, then there is no communication involved. The data can be distributed along the respective dimension and all the data for computation will be in local memory. (Initialize $i \leftarrow 1$).

Step 1: Distribute LHS array along dimension i, *i.e.,* set

$$\vec{c_i}.[T^{-1}] = (1\ 0\ \cdots\ 0).$$

Step 2: Choose an RHS array which does not have a row in the reference matrix the same as that of LHS array. For all j set:

$$\vec{a_j^p}.[T^{-1}] = (0\ 0\ \cdots\ 0\ 1)$$

for all references to that array and $\overline{a_{k \neq j}} \cdot \vec{q_n} = 0$ (where $\vec{q_n}$is the nth column of T^{-1}.

If a valid T^{-1} is found, check the determinant of T^{-1}. If non-zero block transfers are possible for that RHS array, (break) go to Step 3.

If there are no valid T^{-1} or the determinant of T^{-1} is zero for all j, block transfers are not possible on that array with the given distribution of the LHS array; therefore, increment j and go to Step 2.

Step 3: Repeat Step 2 for all the reference matrices of a particular array.

Step 4: Repeat Step 2 for all distinct arrays on RHS. (Increment p)

Step 5: Check the number of arrays where block transfers are possible.

Step 6: Repeat Step 1 to Step 4 for LHS array distributed along all dimensions (Increment i).

Step 7: Compare the number of arrays that can have block transfers and distribute LHS array along the dimension which yields maximum number of block transfers for the arrays on the right hand side.

Figure 1 Algorithm for data distribution and loop transformations

array. There is no communication due to that array, since they are always mapped onto the same processor. If all the references of all the arrays have a row in the data reference matrix identical to that of the LHS array, then the entire loop can be distributed along that dimension and there is no communication.

- If the above condition does not hold, choose the entries in T^{-1} such that the following conditions hold:

  1. some dimension of the RHS reference consists only of the transformed innermost loop index, *e.g.* $A(*, \ldots, j_n, \ldots, *)$; and
  2. all the other dimensions are independent of the innermost loop index (that is, "$*$" indicates a term independent of j_n).

This means the transformed reference matrix must have only one non-zero in some row r, and that non-zero must occur in column n. If this condition is satisfied, then dimension r of the RHS array is not a distributed dimension; thus, we can move communication arising from that RHS reference outside the innermost loop. This allows a block transfer to the local memory before the execution of the innermost loop. This means that a row in the transformed data reference matrix $A'_{\mathcal{R}}$ has a row with all entries zero except in the last column, which is non-zero. Also, the last column of the $A'_{\mathcal{R}}$ has all remaining entries as zero.

- If communication could be moved out of the innermost loop, the previous step can be applied repeatedly staring with the deepest loop outside the innermost and working outward; this process can either stop at some level of the outside which communication can not be moved.

The transformation should also satisfy the condition that the determinant is non-zero and must preserve the dependences in the program.

3.2 The Algorithm

Consider the following loop where n is the loop nesting level and d the dimension of the arrays.

---

for $l_1 = 1$ to N_1 do
...
for $l_n = 1$ to N_n do

$$L\left[\begin{pmatrix} c_{11} & \dots & c_{1n} \\ \ddots & & \\ c_{d1} & \dots & c_{dn} \end{pmatrix}\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} + \begin{pmatrix} b_1^l \\ \vdots \\ b_d^l \end{pmatrix}\right] = R\left[\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \ddots & & \\ a_{d1} & \dots & a_{dn} \end{pmatrix}\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} + \begin{pmatrix} b_1^r \\ \vdots \\ b_d^r \end{pmatrix}\right]$$

Let the inverse of the transformation matrix be

$$T^{-1} = Q = \begin{pmatrix} q_{11} & \dots & q_{1n} \\ \ddots & & \\ q_{n1} & \dots & q_{nn} \end{pmatrix}.$$

The algorithm is shown in Figure 1. We use the notation $A(i,:)$ to refer to the ith row of a matrix A, and $A(:,j)$ to refer to the jth column of a matrix A.

4 Examples

We illustrate the use of the algorithm through several examples in this section. The reader is referred to [9] for a detailed discussion of the algorithm. In the following discussion, we refer to the matrix T^{-1} as the matrix Q.

Example 2: Matrix Multiplication

```
for i = 1 to N do
  for j = 1 to N do
    for k = 1 to N do
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

The reference matrices of the arrays are:

$$C_{\mathcal{R}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

$$A_{\mathcal{R}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \text{and}$$

$$B_{\mathcal{R}} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Step 1: (C distributed along first dimension). Set

$$\begin{aligned} \mathcal{C}_{\mathcal{R}}(\vec{1},:) \cdot Q(\vec{:},1) &= 1 \\ \mathcal{C}_{\mathcal{R}}(\vec{1},:) \cdot Q(\vec{:},2) &= 0 \\ \mathcal{C}_{\mathcal{R}}(\vec{1},:) \cdot Q(\vec{:},3) &= 0 \end{aligned}$$

Therefore we have, $q_{11} = 1$, $q_{12} = 0$, and $q_{13} = 0$.

Step 1a: (Derive distribution of Array A) Since row 1 of A is the same as that of C, *i.e.*, $\mathcal{C}_{\mathcal{R}}(1,:) = \mathcal{A}_{\mathcal{R}}(1,:)$, distribute A and C identically.

Step 2.1: (Derive distribution for Array B) Check if you can find a matrix, $\mathcal{B}_{\mathcal{R}}Q$ of the form

$$\mathcal{B}_{\mathcal{R}}Q = \begin{pmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{pmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} \mathcal{B}_{\mathcal{R}}(\vec{1},:) \cdot Q(\vec{:},1) &= 0 \\ \mathcal{B}_{\mathcal{R}}(\vec{1},:) \cdot Q(\vec{:},2) &= 0 \\ \mathcal{B}_{\mathcal{R}}(\vec{1},:) \cdot Q(\vec{:},3) &= 1 \end{aligned}$$

Therefore we have, $q_{31} = 0$, $q_{32} = 0$, and $q_{33} = 1$. In addition, set $\mathcal{B}_{\mathcal{R}}(\vec{2},:) \cdot Q(\vec{:},3) = 0$. This implies $q_{23} = 0$. Therefore, the first dimension of B is not distributed. Finally we have,

$$T^{-1} = Q = \begin{pmatrix} 1 & 0 & 0 \\ q_{21} & q_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

For a unimodular transformation, $q_{22} = \pm 1$. Note that dependence vector is (0 0 1), and therefore, there are no constraints on q_{21}. This results in the identity matrix as the transformation matrix, and thus nothing need be done. Distribute A and C by rows, and B by columns. The code shown next gives the best performance we can get in terms of parallelism and locality.

```
for u = 1 to N do
  for v = 1 to N do
```

read $B(*, v)$
for $w = 1$ to N
  $C(u, v) = C(u, v) + A(u, w) * B(w, v)$

We go ahead and complete the algorithm by looking at distributing the LHS array in the next dimension.

Step 1.1: (C distributed along second dimension). Set

$$\begin{aligned} \mathcal{C}_{\mathcal{R}}(\vec{2}, :) \quad \cdot \quad Q(\vec{:}, 1) = 1 \\ \mathcal{C}_{\mathcal{R}}(\vec{2}, :) \quad \cdot \quad Q(\vec{:}, 2) = 0 \\ \mathcal{C}_{\mathcal{R}}(\vec{2}, :) \quad \cdot \quad Q(\vec{:}, 3) = 0 \end{aligned}$$

Therefore we have, $q_{21} = 1$, $q_{22} = 0$, and $q_{23} = 0$.

Step 1.1a: (Derive distribution for Array B). Since second row of $\mathcal{B}_{\mathcal{R}}$ is the same as the second row of $\mathcal{C}_{\mathcal{R}}$ distribute B same as C.

Step 2.2: (Derive distribution for Array A). Check if you can find a matrix, $\mathcal{A}_{\mathcal{R}}Q$ of the form

$$\mathcal{A}_{\mathcal{R}}Q = \begin{pmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{pmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} \mathcal{A}_{\mathcal{R}}(\vec{1}, :) \quad \cdot \quad Q(\vec{:}, 1) = 0 \\ \mathcal{A}_{\mathcal{R}}(\vec{1}, :) \quad \cdot \quad Q(\vec{:}, 2) = 0 \\ \mathcal{A}_{\mathcal{R}}(\vec{1}, :) \quad \cdot \quad Q(\vec{:}, 3) = 1 \end{aligned}$$

Therefore we have, $q_{11} = 0$, $q_{12} = 0$ and $q_{13} = 1$; and $\mathcal{A}_{\mathcal{R}}(\vec{2}, :) \cdot Q(\vec{:}, 3) = 0 \Longrightarrow q_{33} = 0$.

Finally we have,

$$T^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ q_{31} & q_{32} & 0 \end{pmatrix}.$$

For a unimodular transformation, $q_{32} = \pm 1$. Therefore,

$$T^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

and

$$T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Distribute the arrays A, B, and C by columns. The transformed loop is given below:

---

for $u = 1$ to N do

for $v = 1$ to N do
  read $A(*, v)$
  for $w = 1$ to N do
    $C(w, u) = C(w, u) + A(w, v) * B(v, u)$

We see that the performance of the loop is similar in both the cases. Therefore the array C can either be distributed by columns with the above transformation, or by rows with no transformation for the same performance with respect to communication. Consider the SYR2K (from BLAS) example shown below.

Example 3: SYR2K
for $i = 1$ to N do
  for $j = i$ to $min(i + 2b - 2, N)$ do
    for $k = max(i - b + 1, j - b + 1, 1)$ to $min(i + b - 1, j + b - 1, N)$ do
      $C[i, j - i + 1] \mathrel{+}= A[k, i - k + b] * B[k, j - k + b] + A[k, j - k + b] * B[k, i - k + b]$

The reference matrices for the arrays are:

$$C_{\mathcal{R}} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{pmatrix},$$

$$A^1_{\mathcal{R}} = B^2_{\mathcal{R}} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \end{pmatrix}, \text{and}$$

$$A^2_{\mathcal{R}} = B^1_{\mathcal{R}} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}.$$

Step 1: (C row distributed) Set

$$\begin{aligned} \mathcal{C}_{\mathcal{R}}\vec{(}1, :) \quad \cdot \quad Q(\vec{:}, 1) = 1 \\ \mathcal{C}_{\mathcal{R}}\vec{(}1, :) \quad \cdot \quad Q(\vec{:}, 2) = 0 \\ \mathcal{C}_{\mathcal{R}}\vec{(}1, :) \quad \cdot \quad Q(\vec{:}, 3) = 0 \end{aligned}$$

Therefore we have, $q_{11} = 1$, $q_{12} = 0$, and $q_{13} = 0$. None of the other references matrices have any row common with $\mathcal{C}_{\mathcal{R}}$.

Step 1.1: (Derive distribution of A for the first reference; check if first dimension of A can be not distributed) Check if you can find a matrix, ${\mathcal{A}_{\mathcal{R}}}^1 Q$ of the form

$${\mathcal{A}_{\mathcal{R}}}^1 Q = \begin{pmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{pmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} {\mathcal{A}_{\mathcal{R}}}^1\vec{(}1, :) \quad \cdot \quad Q(\vec{:}, 1) = 0 \\ {\mathcal{A}_{\mathcal{R}}}^1\vec{(}1, :) \quad \cdot \quad Q(\vec{:}, 2) = 0 \end{aligned}$$

$$\mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}1,:) \quad \cdot \quad Q(\vec{:},3) = 1$$

Therefore, $q_{31} = 0$, $q_{32} = 0$, and $q_{33} = 1$. In addition, $\left(\mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}2,:) \cdot Q(\vec{:},3) = 0\right)$ implies $(q_{13} - q_{33} = 0)$, which is impossible. Therefore, the first dimension of A has to be distributed.

Step 1.2: (Derive distribution of A using first reference; check if second dimension of A can be not distributed) Check if you can find a matrix, $\mathcal{A}_{\mathcal{R}}{}^{1}Q$ of the form

$$\mathcal{A}_{\mathcal{R}}{}^{1}Q = \begin{pmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} \mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}2,:) \quad &\cdot \quad Q(\vec{:},1) = 0 \\ \mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}2,:) \quad &\cdot \quad Q(\vec{:},2) = 0 \\ \mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}2,:) \quad &\cdot \quad Q(\vec{:},3) = 1 \end{aligned}$$

and $\mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}1,:) \cdot Q(\vec{:},3) = 0$. Therefore, $(q_{11} - q_{31} = 0) \Longrightarrow q_{13} = 1$; $(q_{12} - q_{32} = 0) \Longrightarrow q_{32} = 0$; and $(q_{13} - q_{33} = 1) \Longrightarrow q_{33} = -1$, which is impossible since $q_{33} = 1$. Thus the second dimension of A also has to be distributed. Based on an analysis of the first reference of A, every dimension of A must be distributed. A similar result follows from an analysis of the second reference to A as well. Since the reference matrix for array A and B are the same, there can be no block transfers for B as well.

Step 2.0: (C column distributed) Set

$$\begin{aligned} \mathcal{C}_{\mathcal{R}}\vec{(}2,:) \quad &\cdot \quad Q(\vec{:},1) = 1 \\ \mathcal{C}_{\mathcal{R}}\vec{(}2,:) \quad &\cdot \quad Q(\vec{:},2) = 0 \\ \mathcal{C}_{\mathcal{R}}\vec{(}2,:) \quad &\cdot \quad Q(\vec{:},3) = 0 \end{aligned}$$

Therefore, $(-q_{11} + q_{21} = 1)$ implies $(q_{11} = q_{21} - 1)$, $(-q_{12} + q_{22} = 0)$ implies $(q_{12} = q_{22})$, and $(-q_{13} + q_{23} = 0)$ implies $(q_{13} = q_{23})$.

Step 2.1a: (Derive distribution of A; check if the first dimension of A can be not distributed) Set

$$\begin{aligned} \mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}1,:) \quad &\cdot \quad Q(\vec{:},1) = 0 \\ \mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}1,:) \quad &\cdot \quad Q(\vec{:},2) = 0 \\ \mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}1,:) \quad &\cdot \quad Q(\vec{:},3) = 1 \end{aligned}$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$, and $\left(\mathcal{A}_{\mathcal{R}}{}^{1}\vec{(}2,:) \cdot Q(\vec{:},3) = 0\right) \Longrightarrow (q_{13} - q_{33} = 0) \Longrightarrow$ $q_{13} = 1$ and $q_{23} = 1$.

Step 2.1b: (Second reference of A, check if the second reference allows the first dimension of A to be not distributed): Set

$$\begin{aligned} \mathcal{A}_{\mathcal{R}}{}^{2}\vec{(}1,:) \quad &\cdot \quad Q(\vec{:},1) = 0 \\ \mathcal{A}_{\mathcal{R}}{}^{2}\vec{(}1,:) \quad &\cdot \quad Q(\vec{:},2) = 0 \end{aligned}$$

$$\mathcal{A}_{\mathcal{R}}{}^{2}\vec{(}1,:) \quad \cdot \quad Q(\vec{:},3) = 1$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$, and $\left(\mathcal{A}_{\mathcal{R}}{}^{2}\vec{(}2,:) \cdot Q(\vec{:},3) = 0\right) \Longrightarrow (q_{23} - q_{33} = 0)$ and $q_{23} = 1$.

Thus both references to A allow A to be not distributed by its first dimension. Thus, A can be column distributed (by its second dimension. Since B has identical array reference matrices as those of A, the array B can also be distributed by columns. Recall, that we started out with a column distribution of C. Thus, we have the inverse of the transformation matrix as

$$T^{-1} = \begin{pmatrix} q_{11} & q_{12} & 1 \\ q_{21} & q_{22} & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

We choose the unknown values such that T is a legal unimodular transformation. A possible T^{-1} is

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$T = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix}.$$

The transformed reference matrices are as follows:

$$C'_{\mathcal{R}} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix},$$

$$A'^{,1}_{\mathcal{R}} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \text{and}$$

$$A'^{,2}_{\mathcal{R}} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Using the above algorithm we distribute the arrays A, B, and C by columns and the transformed code with block transfers is as follows:

```
for u = 1 to N do
  for v = u to min(u + 2b − 2, N) do
    read A(∗, v + b); read A(∗, u + v + b); read B(∗, u + v + b); read B(∗, v + b);
    for w = max(u − b + 1, v − b + 1, 1) to min(u + b − 1, v + b − 1, N) do
      C[v + w + 1, u]+ = A[w, v + b] ∗ B[w, u + v + b] + A[w, u + v + b] ∗ B[w, v + b]
```

5 Summary

This paper illustrated an algorithm which derives the terms in the transformation matrix which gives the best locality and minimum communication on distributed memory machines. We used the concept of *data reference matrices* for individual array references. Using this as the starting point,

we systematically derived the best set of transformation matrices which give both good locality while enabling parallelism. Unlike [7], where a *padding matrix* is used along with an arbitrary set of rows in the *basis matrix,* we generate a transformation matrix systematically. The algorithm also gives an optimal distribution of arrays on to the processors such that block transfers are enabled to reduce inter-processor communication. Here, distribution of data only along one dimension is considered. However complex distributions with more than one distributed dimension can be derived using a simple extension of the above algorithm. Work is in progress in deriving more complex distribution of data and iterations along with tiling.

References

[1] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN'93 PLDI,* June 1993, pp. 112–125.

[2] S. Chatterjee, J. Gilbert, R. Schreiber and S. Teng. Optimal evaluation of array expressions on massively parallel machines. RIACS Technical Report TR 92.17.

[3] M. Gupta. *Automatic data partitioning on distributed memory multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1992.

[4] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive environment. Tech. Rep. CRPC TR91-205, Rice University, April 1991.

[5] K. Knobe, J. Lukas and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2), Feb. 1990, 102–118.

[6] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*,13(2) Oct. 1991, 213–221.

[7] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Proc. 5th Workshop on Languages and Compilers for Parallel Computing,* August 1992.

[8] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, Oct. 1991.

[9] J. Ramanujam and A. Narayan. Automatic array distribution and loop transformations. Technical Report TR-94-07, Louisiana State University, January 1994.

[10] S. Wholey. *Automatic data mapping for distributed-memory parallel computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991.

A Novel Approach Towards Automatic Data Distribution

Jordi Garcia, Eduard Ayguade, and Jesus Labarta
Universitat Politecnica de Catalunya, Spain

In this paper we present a novel approach to automatically perform data distribution. All the constraints related to parallelism and data movement are contained in a single data structure, the Communication-Parallelism Graph (CPG). The problem is solved using a linear 0-1 integer programming model and solver. This allows us to find the best solution for the problem for one-dimensional array distributions. An iterative approach is used to solve the multi-dimensional problem. We also show the feasibility of using this approach to solve the problem in terms of compilation time and quality of the solutions generated.

A Novel Approach Towards Automatic Data Distribution

Jordi Garcia, Eduard Ayguadé and Jesús Labarta

Computer Architecture Department, Universitat Politècnica de Catalunya
Gran Capità s/núm, Mòdul D6, 08034 - Barcelona, Spain
{jordig | eduard | jesus}@ac.upc.es

Abstract

Data distribution is one of the key aspects that a parallelizing compiler for a distributed memory architecture should consider, in order to get efficiency from the system. The cost of accessing local and remote data can be one or several orders of magnitude different, and this can dramatically affect performance. In this paper, we present a novel approach to automatically perform static data distribution. All the constraints related to parallelism and data movement are contained in a single data structure, the Communication-Parallelism Graph (CPG). The problem is solved using a linear 0-1 integer programming model and solver. This allows us to find the best solution for the problem for one-dimensional array distributions. The solution is static in the sense that the layout of the arrays does not change during the executionn of the program. An iterative approach is used to solve the multi-dimensional problem. We also show the feasibility of using this approach to solve the problem in terms of compilation time and quality of the solutions generated.

1 Introduction

Data distribution is one of the key aspects that have to be considered in a parallelizing environment for Massive Parallel Machines, in which each processor has direct access to a local (or close) memory and indirect access to the remote memories of other processors. The cost of accessing a local memory location can be more than one order of magnitude faster than the cost of accessing a remote memory location. In these systems, the choice of a good data distribution can dramatically affect performance because of the non-uniformity of the memory system. However, the data distribution problem has been proved to be NP-complete in [Krem94].

Two different alternatives are currently offered to the user to address this problem. The first one is to give him the responsability of deciding how data structures have to be aligned each other, and distributed among the memory modules of the system. In this case, the sequential code is annotated with directives and executable statements, offered by current languages such as

Fortran D [FHKK90], High Performance Fortran [HPFF93], or Vienna Fortran [ZBCM91]. The second alternative is to leave this task to the compiler. In any case, classical aspects such as data movement, parallelism, and load balance have to be taken into consideration in a unified way to efficiently solve the data distribution problem.

Once the data distribution strategy has been chosen, a sophisticated compiler must take care of the explicit management of local name spaces, and generation of a message-passing single-program multiple-data code for a given distributed-memory target machine, where all processors execute the same program, but operate on distinct data items. The basic compilation rule used by most of the compilers is the owner computes rule, where the processor that owns a data item is the one that performs all the computations to update it.

Several researchers have targeted their research efforts to this topic. For instance, the Crystal compiler and language project [LiCh90], the implementation of PARADIGM [Gupt92] on top of Parafrase-2 and its continuation on the PTRAN II compiler [GMSS93], the compiler for the ALEXI language [Whol92], the framework for the automatic determination of array alignment and distribution presented in [CGST93, CGSS94], or the automatic data layout strategy [BiKK94] for use in the D programming environment.

All these automatic or semi-automatic data distribution methods perform the job in two main independent steps: alignment and distribution. The alignment step tries to find appropriate alignments between all arrays in a block of code, that is, to decide for each array the dimensions that will be aligned into the dimensions of another array called the template (interdimensional alignment), and for each aligned dimension, to decide whether it is better to shift the array with respect to the template or not (intradimensional alignment). A good alignment will minimize the overhead of interprocessor communication. This goal could be trivially satisfied assigning all the data to one processor, which minimizes the communication overhead. The distribution step decides which dimension or dimensions of the template are distributed, and the number of processors assigned to each of them. The mapping of the arrays is determined by their alignment with respect to the template and its distribution. A good distribution maximizes the potential parallelism of the code, and offers the possibility of further reducing communication by serializing. This goal could be trivially satisfied assigning a datum to each processor, which maximizes parallelism. The distribution step depends on the alignment one, that is, once all arrays have been aligned, then they have to be distributed. But there is a trade-off between communication and parallelism. The layout that achieves the best performance could be one with a non optimal alignment. In other words, to obtain the alignment strategy, some information about distribution should be considered.

Our approach builds the Communication-Parallelism Graph (CPG), a directed weighted graph that holds information about both the data movement and parallelism inherent in a block of

code. This information is used to formulate a minimal path problem with a set of additional constraints, which is solved using a general purpose linear 0-1 integer programming solver. This solver finds the optimal solution for both problems in a single step and in a small amount of time. This allows us to minimize communication while maximizing parallelism, avoiding the use of heuristics and possibly wrong assumptions about distribution during the alignment phase. In addition, the paper shows the feasibility of solving the problem using linear 0-1 integer programming, which allows us to obtain more accurate solutions than other proposals. In the framework of automatic data distribution, the use of linear 0-1 integer programming solvers was proposed by [BiKK94] when looking for a dynamic solution for the data layout problem. This paper presents the method that finds the optimal solution for one-dimensional array distributions. The extension to multi-dimensional distributions is also outlined.

Figure 1 shows the code that will be used as working example along the paper, and the corresponding HPF [HPFF93] data mapping directives that are selected by our approach, after performing the corresponding analysis and finding the minimal cost solution. The parallelization directive in loop j has been specified using the syntax defined in [APR94].

```
CHPF$ DECOMPOSITION TEMPLATE(N, N, N)
CHPF$ ALIGN A(I,J,K) WITH TEMPLATE(K,J,I)
CHPF$ ALIGN B(I,J,K) WITH TEMPLATE(I,J,K)
CHPF$ ALIGN C(I,J) WITH TEMPLATE(1,I,J)
CHPF$ ALIGN D(I,J,K) WITH TEMPLATE(I,J,K)
CHPF$ DISTRIBUTE TEMPLATE(:,BLOCK,:)

  do i = 1, N
CAPR$ DO PAR ON B<:,1~1,:>
     do j = 1, N
        do k = 1, N
           B (i, j, k) = C (j, k) + i
           A (k, j, i) = B (i, j, k) + 1
        enddo
        do k = 2, N
           D (i, j, k) = D (i, j, k - 1)
        enddo
     enddo
  enddo
```

Figure 1: Working example with data mapping directives.

The rest of this paper is organized as follows: in Section 2 we describe the Communication-Parallelism Graph (CPG), the main structure of our approach. Section 3 shows how the constraints in the CPG can be formulated as a linear 0-1 integer programming problem, when a single array dimension is distributed. Section 4 gives some ideas about the extension of this work to multi-dimensional distributions. Section 5 summarizes our implementation and shows our first experimental results. Section 6 outlines some related work with special emphasis in the differences between their and our approaches. And finally, a few concluding remarks are given in Section 7.

2 The Communication-Parallelism Graph

The main structure of our method is the Communication-Parallelism Graph (CPG). It is a directed graph that contains all the information related to communication and parallelism in a block of code. The CPG is created from the analysis of all assignment statements within loops that contain one array in the left hand side of the assignment.

2.1 CPG Nodes

The nodes in the CPG are organized in columns. Each column represents an array, and it contains as many nodes as the maximum dimensionality d of all arrays in the block of code. Each node in a column, thus, represents one dimension of the array that will be mapped into one of the dimensions of a common array called template with dimensionality d. If the array has dimensionality d' < d, then the column is padded with (d - d') additional nodes. These nodes are included to allow an embedding of the array on the template (sequentialization). As seen in the example in Figure 1, array C is embedded into the first dimension of the template. Since four arrays are used in our working example (three of them with dimensionality three, and one with dimensionality two) the CPG consists of 12 nodes (four columns with three nodes each). This is the basis of the CPG, over which the communication and parallelism information will be added in terms of data movement edges and parallelism hyperedges.

2.2 CPG Data Movement Edges

Data movement information is obtained from the analysis of reference patterns. The meaning of reference patterns is defined in [LiCh90], and represents a collection of dependences between arrays in both sides of an assignment statement. When the same array is used in both sides, the reference pattern is called a self-reference pattern. For instance, consider the code example of Figure 1. From the array assignment statements inside the loop nest, two reference patterns can be extracted:

```
B (i, j, k) ← C (j, k)
A (k, j, i) ← B (i, j, k)
```

and one self-reference pattern:

```
D (i, j, k) ← D (i, j, k - 1)
```

For each reference pattern between two different arrays, (d * d) directed edges are added, where d is the maximum dimensionality of all arrays. The edges connect all nodes of the array to be read (right hand side) to all nodes of the array to be updated (left hand side), and represent all alignment possibilities between these two arrays. The weight that is assigned to the edge is a symbolic expression that represents the cost of the data movement that has to be performed when these two dimensions are aligned each other and distributed. This cost reflects the number of

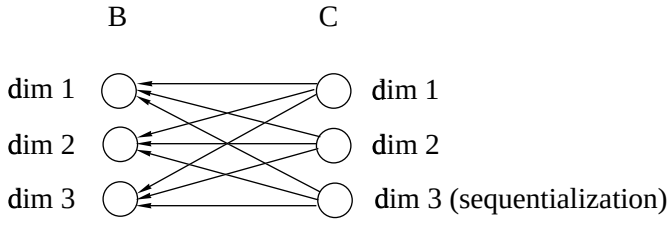
remote memory accesses that have to be performed, and it is a function of the number of processors that will be assigned to this dimension, and the communication rate between two processors. When a self reference pattern is found, d self edges are added in the column of the referenced array, one in each node. As in the previous case, the weight is a symbolic expression that represents the cost of the data movement that has to be performed when this dimension is distributed. Several edges between a pair of nodes are replaced by a single edge with a weight equal to the sum of the original ones.

In order to estimate data movement costs, reference patterns are matched with a set of predefined data movement routines. The routines that have been used are listed in Table 1, together with the reference patterns that matches them. All the information in this table is machine dependent and should be tailored to the specific target machine [LiCh91].

Pattern	**Primitive**
$i_p \cong j_p$	LOCAL MEMORY ACCESS
const $(i_p - j_p)$	ONE-TO-ONE
const (i_p)	MANY-TO-ONE
const (j_p)	ONE-TO-MANY
$i_p \neq j_p$	MANY-TO-MANY

Table 1: Matching between reference pattern and communication primitive.

In the example of Figure 1, when analyzing the first assignment statement, the following edges between arrays B and C are inserted:



The communication primitive that is assigned as weight in the edge that connects the first dimension of array C to the second dimension of the array B (C[1]-B[2] for shorter) is a *local memory access*. This means that if these two dimensions are aligned and distributed, when loading the elements of array C to update array B all the memory accesses will be local. The same happens if C[2]-B[3] are aligned. The cost of a *many-to-many* data movement will be assigned to the edge C[1]-B[1]. This means that if these two dimensions are aligned and distributed, a data movement involving all processors will be performed while executing the loop. The same happens if C[1]-B[3], C[2]-B[1], or C[2]-B[2] are aligned. And the cost of a *one-to-many* primitive will be

assigned to the edges C[3]-B[1], C[3]-B[2] and C[3]-B[3]. Distributing the third dimension of array C means to sequentiallize it. So the processor owner of array C will have to send the whole one

A similar set of edges and data movement information is added when analyzing the assignment of B to array A. Finally, from the last assignment, a self-reference pattern is extracted. The data movement edges added are self-edges. The communication primitive that is assigned as weight to the self edge of the first and second dimensions of array D, is a *local memory access*. A *one-to-one* data movement is assigned to the self-reference pattern D[3]-D[3].

After adding the data movement information, the CPG (without weights) that is obtained is shown in Figure 2.a. In this graph, the edges due to the assignment of array B to array A, the assignment of array C to array B, and the self assignment of array D can be seen. The cost functions have not been represented.

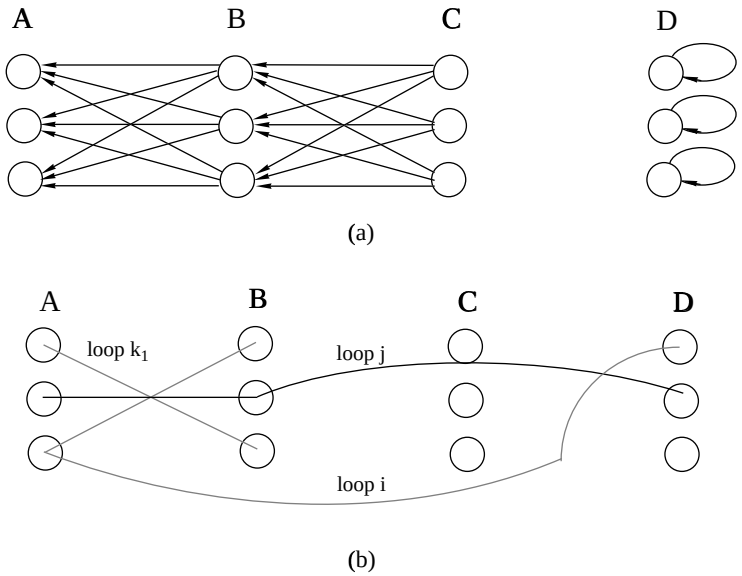


Figure 2: CPG with (a) data movement edges and (b) parallelism hyperedges.

Note that the CPG, when only filled with the data movement information, is not the Component Affinity Graph (CAG) used by other authors [LiCh90, Gupt92]. The meaning of the edges in the CAG is a preference for alignment, that is, how good is to align two dimensions. The weight should reflect the extra communication cost incurred if those dimensions are not aligned. The problem is that one can not identify two unique alignment configurations under which the communication costs may be estimated and compared to determine the penalty. In our approach, the meaning of one edge in the CPG is just the opposed, that is, how costly is (in terms of data movement) to align and distribute the two dimensions. This cost is independent of any other alignment strategy, and it will be useful to determine the distribution strategy.

2.3 CPG Parallelism Hyperedges

Parallelism information is obtained from data dependence analysis. First, all loops that can be executed in parallel are detected. In distributed memory machines you can fully parallelize one loop when the loop does not carry any data flow dependence [Tsen93]. When all possible parallel loops have been marked, the analyzer must only look at the left hand side of the assignments that are inside any parallel loop. According to the owner computes rule, the processor that owns a data element is the one who performs all the computations to update it. So if one loop is going to be parallelized, it must be ensured for the arrays that appear in the left hand side of each assignment statement inside the loop, that the dimension subscripted by the loop control variable of the parallel loop is the one that will be distributed.

Concretelly, for each parallel loop, the analyzer searches for all the assignment statements inside it with an array in the left hand side. If the array uses the loop control variable in any dimension of its subscript, then the analyzer links the corresponding node to a hyperedge. The hyperedge is a generalization of an edge, as it can connect more than two nodes, and in this case, it is undirected. Each parallel loop has a hyperedge in the CPG. If all the nodes connected by a hyperedge are aligned and distributed, then the corresponding loop can be parallelized. But if one of the nodes connected by a hyperedge is not aligned, then the loop can not be parallelized, and guard expressions must be inserted before each assignment in that loop [Tsen93].

The weight of a hyperedge represents the execution time that is saved when the loop is parallelized. This time is a function of the sequential execution time of the loop, the number of processors that will be assigned to that dimension and the overhead due to the parallel execution itself.

In the example of Figure 1, after performing the data dependence analysis, one can see that there is a single flow dependence carried by the second k-loop. This means that the i, j, and the first k loops can be executed in parallel. So the parallelism information that the analyzer will add to the CPG is shown in Figure 2.b. Three hyperedges have been added: one linking B[1], A[3], and D[1]; this one is associated to loop i. Another hyperedge links A[2], B[2] and D[2], and it is associated to loop j. The last one links A[1] and B[3], and it is associated to the first loop k.

2.4 Problem Formulation

Once the CPG is built, it contains all the necessary information regarding data movement and parallelism in the block of code analyzed. The weights in the CPG are expressed in a symbolic form, function of the number of processors that will be assigned to that dimension. If all symbolic expressions in the CPG are replaced for its constant value assuming P processors, then the weights in the edges will represent the cost of aligning and distributing the corresponding dimensions. In this section, the CPG can be considered as an undirected graph, and all pairs of edges connecting

two nodes in different direction, can be replaced by a single undirected edge with weight sum of the original edges. So if one node (dimension) of each column (array) is selected, the sum of the weights of all data movement edges that connect any two nodes inside the selection is the total data movement cost of the block of code when distributing the selected dimensions. Similarly, the sum of weights of all parallelism hyperedges whose nodes remain completely inside the selected set is the total execution time saved when parallelizing and assigning P processors.

It can be assumed that there exists a path between any pair of columns in the CPG. If any set of arrays is not connected, then this set will be analyzed independently, and assigned a different template. The reason is that a relation (alignment) between two unrelated sets of arrays should not be imposed. If an array is connected to the CPG but only through a hyperedge, then all nodes of this array will be connected to all nodes of any other array with a data movement edge, and assigned a null weight.

Now, it can be ensured that the CPG is an undirected connected graph, where every pair of columns are connected by a path, and all edges and hyperedges have constant weights. The problem to solve is to find the set of nodes (one for each column), that minimizes the total execution time. The total execution time can be computed as the sequential execution time plus the cost in time due to data distribution and parallelization; according to the weights in the CPG, this cost is the data movement overhead minus the time saved due to the parallel execution of the loops. The problem is formulated as a linear 0-1 integer programming problem, where the objective function to minimize is the total execution time.

3 One-dimensional Distribution

Linear integer programming is a tool for solving optimization problems. As stated by [BiKK94] data layout problems can be very efficiently solved using linear integer programming. In this case, the problem to solve is to find a path in the CPG that includes exactly one node of each column, so that the sum of weights of the edges (data movement edges) minus the sum of weights of the hyperedges (parallelism hyperedges) that connect nodes inside the selected path, is minimized. This problem can be formulated as a linear 0-1 integer programming problem, that is, a linear integer programming problem where each variable has two possible values: 0 or 1.

The model that we have selected is a shortest path problem with a set of additional constraints. When formulating it, two dummy nodes are considered: a source S and a sink T. All edges going from the dummy source S to each node in the first column, plus all edges going from each node in the last column to the dummy sink T, must be defined as well. In addition, all columns C with self-edges will be replaced by two columns named C and C', and each node in column C will be linked by an edge to each node in column C'. The weights of the new edges will be the same

of the self-edge if it links two nodes of the same level, or infinite otherwise. This infinite weight prevents from selecting two nodes of different level from the same column. Figure 3 shows the graph that finally will be considered to solve the problem for the working example of Figure 1. Neither infinite weighted edges are present, nor hyperedges.

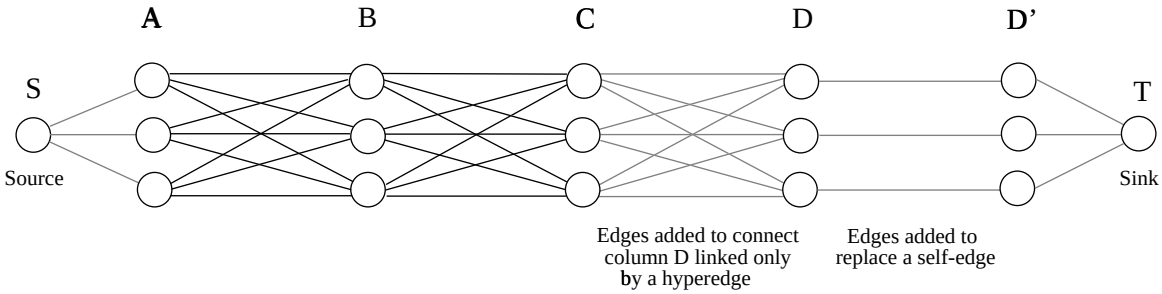


Figure 3: CPG with all the data movement edges that fulfils the properties mentioned in Section 3.

As it is usual in this kind of problems, one 0-1 integer variable associated to each edge, plus one 0-1 integer variable associated to each hyperedge are defined.

Let Y_{PQ} denote the set of edges connecting nodes in column P to nodes in column Q. When Y_{PQ} is not empty, it contains (d x d) elements. If S denotes the column associated to the source, the sets Y_{SP} are all empty, except when P denotes the first of the columns, and it has d elements. Similarly, if T denotes the column associated to the sink, the sets Y_{QT} are all empty except when Q denotes the last of the columns, and it has d elements. Let $Y_{PQ}[i, j]$ be the variable associated to the edge connecting node i in column P to node j in column Q. Its value will be one if the corresponding edge belongs to the path, and zero otherwise. Note that, as the graph is undirected, $Y_{PQ}[i, j]$ is the same than $Y_{QP}[j, i]$. Finally, if an index is assigned to each hyperedge, Z_m will denote the 0-1 integer variable associated to the m-th hyperedge. Similarly, its value will be one if the nodes it links belong to the path, and zero otherwise.

To ensure the correctness of the solution, some constraints must be defined. There are four types of constraints:

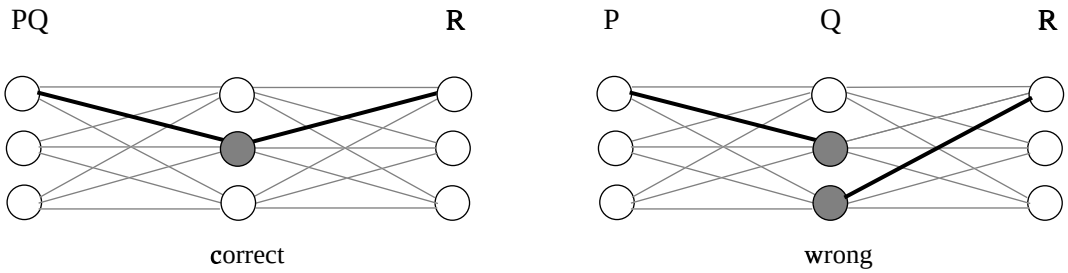
C1: The solution is a path.

C2: The path goes through one node in each column.

C3: All edges connecting selected nodes, must be included as well.

C4: If a node of a hyperedge is not selected, neither it is.

Constraints C1 ensure that the solution is connected. That is, for each column Q conected to more than one column P and R, if one edge leading to a node in Q is selected in the set Y_{PQ}, one edge leaving this same node must be selected in the set Y_{QR}. This can be graphically seen in the left hand side of the next figure. In the right hand side there is a wrong case:



In terms of the variables and their values, it can be stated that at each node of each column Q (except S and T) connected to more than one column P and R, the sum of the values of the edges that connect this node to column P, must be equal to the sum of the values of the edges that connect this node to column R. This must be accomplished for each pair of sets Y_{PQ}-Y_{QR} with a column in common.

$$\sum_{j=1}^{ncols} Y_{PQ}[j,i] = \sum_{j=1}^{ncols} Y_{QR}[i,j] \qquad ;i = 1 \ldots ncols$$

Constraints C2 and C3 can be specified together. They ensure that only one node can be selected in each column, and that all edges connecting two selected nodes, must also be included. These can be accomplished forcing that, for each non empty set of edges Y_{PQ}, exactly one edge must be selected. An example of this constraint can be seen in the left hand side of the next figure. On the right hand side there is an example where the selection is wrong:

This can be stated, in terms of variables and their values, that the summatory of a non empty set of edges Y_{PQ} must equal one. This set of constraints, must be accomplished for each non empty set of edges.

$$\sum_{i=1}^{ncols} \sum_{j=1}^{ncols} Y_{PQ}[i,j] = 1 \qquad ;PQ = 1 \ldots nsets$$

As a consequence of these constraints, the path found in the solution can be not simple. This means that it could contain cycles, or that the same node could be more than once in the path. Note that this is not contradictory with the 2nd constraint, since exactly one different node in each column belongs to the path.

Finally, constraints C4 ensure the correct behavior of the hyperedges. A hyperedge m belongs to the selection if all nodes linked by it have been selected. One node n of column P belongs to the selection, if the value of one of the edges that connects it to any other column Q equals one. Or in other words, if the summatory of the values of the edges that connect it to column Q equals one. This can be stated, in terms of variables and their values, that the sum of the values of the edges that connect the node n of column P to any other column Q, is greater or equal than Z_m, where Z_m is the 0-1 integer variable associated to the m-th hyperedge.

$$\sum_{j=1}^{ncols} Y_{PQ}(n, j) \geq Z_m$$

This must be accomplished for each node linked by the hyperedge. For instance, in the CPG corresponding to the example code of Figure 1, the hyperedge labeled i links nodes B[1], A[3] and D[1]. Column A is connected to column B, column B is connected to column C, and column C is connected to column D. This set of constraints, can be stated as:

$$\sum_{j=1}^{ncols} Y_{BC}(1, j) \geq Zi \qquad ; \sum_{j=1}^{ncols} Y_{AB}(3, j) \geq Z_i \qquad ; \sum_{i=1}^{ncols} Y_{CD}(i, 1) \geq Z_i$$

The 0-1 integer variable associated to hyperedge labeled i is Z_i. In this case, if any of the three summatories are zero, then Z_i will also be zero. This constraint must be formulated for each hyperedge m in the graph.

Once all constraints have been formulated, the objective function to minimize must be specified: the sum of the weights of all selected edges, minus the sum of the weights of all selected hyperedges. Let p be the total number edges in the CPG. The sum of the weights of all selected edges can be expressed as the scalar product in the space R^p of Y and C:

$$CostOfEdges = Y \cdot C$$

where Y represents the vector of all 0-1 integer variables associated to edges, and C represents their respective weights.

Similarly, let m be the number of hyperedges in the CPG. The sum of the weights of all selected hyperedges can be expressed as the scalar product in the space R^m of vectors Z and T:

$$CostOfHyper = Z \cdot T$$

where Z represents the vector of all 0-1 integer variables associated to hyperedges, and T represents their respective weights.

Finally, the objective function can be expressed as:

$$Cost = CostOfEdges - CostOfHyper$$

The objective function must be minimized. When all constraints and the objective function have been specified, the integer linear programming solver finds the optimal solution (minimum) subject to the specified constraints.

In the Appendix, the full description of the constraints set for the working example of Figure 1 can be found, using the syntax specified by the LINGO optimization modeling language [LIND94].

4 Multi-dimensional Distribution

This section introduces some ideas to adapt this framework to the multi-dimensional problem. The introduction will be made in two steps. In the first one we will assume that the number of processors that is assigned to each dimension is known. In the second step, the problem will be to find the combination of processors that achieves the best performance.

4.1 The Number of Processors is Known

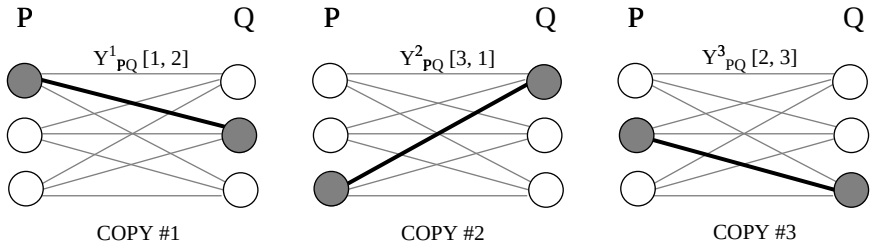
In this step, it is assumed that the number of processors that will be assigned to each dimension is known. This means that if the template is a d-dimensional array, then a set (N_1, N_2, ..., N_d) is known beforehand, where (N_1 x N_2 x ... x N_d) = P, the number of available processors. This general case includes the trivial one, when (N_1, N_2, ..., N_d) is the set (P, 1, ..., 1). Data movement and parallelism cost expressions are function of the number of processors assigned to each dimension, and the block size. In the worst case, all N_k are different each other, so d different constant costs must be computed. To do this, the original CPG is replicated d times, and the cost expression in the k-th copy replaced for its corresponding constant value assuming N_k processors.

Now the problem is to find a path in each copy of the CPG that includes exactly one node in each column, with the additional restriction that paths can not have any node in common. The sum of the values in the data movement edges minus the sum of the values in the parallelism hyperedges that remain inside all paths must be minimized. This is also modeled as a linear 0-1 integer programming problem. Similarly, one 0-1 integer variable is associated to each edge, and one 0-1 variable is associated to each hyperedge. Now, each variable has a new index k, specifying the copy of the CPG it corresponds to. In this case, let $Y^k_{PQ}[i, j]$ be the variable associated to the edge that connects node i of column P to node j of column Q, in the k-th copy of the CPG. The same notation is used for the hyperedges, where Z^k_m represents the m-th hyperedge in the k-th copy of the CPG.

In order to ensure the correctness of the solution, the sets of constraints of the previous section must be specified for each copy of the CPG, and an additional constraint must be added:

C5: Different copies of the same node can not be in different paths.

This set of constraints ensures that paths will not have nodes in common. That is, for each non empty set of edges Y_{PQ}, the number of selected edges that enters to each node and the number of selected edges that leave from each node in all copies must be one. An example of a valid selection of a non empty set of edges Y_{PQ} can be seen in the next figure:



In terms of variables and their values, it can be stated that at each node i of column P, the summatory of values of the edges that connect different copies of this node to a non empty column Q must be 1. And at each node of column Q, the summatory of values of the edges that connect different copies of this node to column P must also be 1:

$$\sum_{k=1}^{ncopies} \sum_{j=1}^{ncols} Y_{PQ}^{k}(i,j) = 1 \qquad ;i = 1 \ldots ncols$$

$$\sum_{k=1}^{ncopies} \sum_{i=1}^{ncols} Y_{PQ}^{k}(i,j) = 1 \qquad ;j = 1 \ldots ncols$$

This model can be simplified in some cases. If the same number of processors is assigned to several template dimensions, a single copy of the CPG for them is enough, in which case two different paths must be selected in this copy. Another simplification is to eliminate any copy k of the CPG when its associated N_k is 1. In this case, as the corresponding dimension is not distributed but it is sequentialized, its cost does not affect the overall performance. In both cases, the set of constraints must be slightly modified.

The problem that one can find with this formulation of the problem is that the weights in the different copies of the CPG are not related. For instance, the execution time of a parallel loop is its sequential time divided by the number of processors assigned to that loop. But if this loop has a nested one, and it is also parallelized, then the execution time of the outer loop is its sequential time assuming that the inner loop has been parallelized, divided by the number of processors. This means that there is a relation between costs of the paths selected in each copy. We are currently working in this direction to have a more accurate model.

4.2 Finding the Number of Processors

In this section we outline two different alternatives to the problem of finding the number of processors that should be assigned to each dimension of the template. The most naive alyternative

would be to generate iterativelly all different sets (N_1, N_2, ..., N_d) of processors, where (N_1 x N_2 x ... x N_d) equals the number of processors P; for each combination solve the 0-1 problem and finally choose the best combination.

The second alternative tries to solve the problem in a single step. If P is a power of 2, then log P different number of processors power of 2 (different than $2^0 = 1$) could be assigned to any dimension. This is the number of copies of the CPG that we require. Cost expressions in the k-th copy are replaced by its constant value assuming 2^k processors, for each k = 1 ... (log P). Some additional constraints have to be added to ensure that only d copies out of the (log P) copies of the CPG are selected, and that these copies belong to a valid set (N_1, N_2, ..., N_d) of processors such that (N_1 x N_2 x ... x N_d) = P.

For instance, if the number of processors P equals 8, and the dimensionality of the template is 3, then all valid sets of processors are: (8, 1, 1), (4, 2, 1), and (2, 2, 2). In this case 3 copies of the CPG are required, the first one with 2 processors, the second one with 4 processors, and the third one with 8 processors. The set of additional constraints, must ensure that:

- If one of the selected paths belong to the third copy (8 processors), then no other path can be selected in any other copy.
- If one of the selected paths belong to the second copy (4 processors), then one more path in the first copy (2 processors) must be selected.
- Otherwise, three paths must be selected in the first copy.

5 Some Experimental Results

The main elements of our tool are described next. The parsing of the code is performed using the parser module of DDT [AGGL94], a tool to analyze reference patterns in Fortran programs, and built on top of ParaScope [KeKT90], an interactive parallel programming environment. DDT obtains all reference patterns after performing some well known optimizations, like expression substitution, subscript substitution, and induction variable detection that improve the quality and quantity of reference patterns analyzed. A profile of the sequential execution is used to estimate the costs of the parallel hyperedges in the CPG. Machine specific information is used to estimate the costs of the data movement edges. The analyzer creates a data file with the set of constraints and the objective function to minimize. This file is the input of a general purpose linear programming solver, which minimizes the objective function, and generates an output file with the final value for each 0-1 integer variable. The general purpose solver used is LINGO [LIND94].

The solution file is used to annotate the original sequential Fortran file with the data mapping and loop parallelization directives. This file is compiled using the xHPF [APR94] compiler form APR Inc, which generates a message-passing Fortran parallelized program. Finally, the Forge

Performance Simulator is used to predict performance, and to validate the solutions generated. All parallel simulations have been done assuming 8 processors.

The set of programs and routines selected[1] to measure this proposal is listed in Table 2. Routine *eflux* has been extracted from the FLO52 program in the Perfect Club, and *tred2* is a routine taken from the Eispack library. These have been analyzed in [Gupt92]. *rhs* is the more time consuming routine in the APPSP NAS benchmark, included in the xHPF benchmark set. From program BARO, also in the xHPF benchmark set, routines *intba1* and *comp* have been extracted. And finally, the *erlebacher* benchmark is a 3-dimensional tridiagonal solver that uses Alternating Direction Implicit integration (also used in [BiKK94] to illustrate their dynamic layout techniques and selected here because of its complexity).

Name	# lines	# parallel loops	# patterns	# arrays	Max Dim
eflux	58	11	161 (4)	5	3
tred2	96	11	38 (8)	4	2
rhs	353	37	346 (5)	4	4
intba1	71	10	30 (15)	10	2
comp	174	18	162 (43)	24	2
erlebacher	288	72	153 (56)	25	3

Table 2: Characteristics of the selected programs.

The static characteristics of the set of programs analyzed are summarized in Table 2. This includes the number of valid code lines in each program (this is, the number of lines once all comment and null lines have been removed), the number of loops that can be executed in parallel in our model of architecture (this is, loops without any loop carried flow dependence), the total number of reference patterns (in brackets the number of reference patterns between different pairs of arrays), the number of arrays used in the programs, and finally the maximal dimensionality.

The complexity of the method is a function of the 0-1 integer variables required in the model, and this is a function of the number of different reference patterns, the maximal dimensionality of the arrays, and the number of possible parallel loops in the program. Table 3 summarizes all this characterictics, and the time spent in finding the optimal solution in a Sun SuperSparc 20.

As a summary, one can see that the analysis of program *erlebacher* is the one that requires more time, as it is also the one that defines more number of 0-1 integer variables. Routine *comp* spends 1.3 seconds, and all other analyses spend less than one second.

1. If the programs selected contain routine calls, these have been inlined. And if the code selected is a routine, it has to be transformed into a program, replacing parameters and arguments by local variables.

Name	# 0-1 variables	# constraints	time
eflux	47	24	0.2 secs
tred2	43	31	0.3 secs
rhs	117	63	0.5 secs
intba1	70	73	0.5 secs
comp	190	180	1.3 secs
erlebacher	576	318	3.4 secs

Table 3: Complexity of the selected programs.

Finally, Table 4 summarizes the run-time behavior of the selected programs. In the second column the sequential execution time is listed. The third column reflects the simulated execution time, when parallelizing the programs and assuming a default mapping. The default mapping for each program is the distribution of the first dimension of all arrays, except for routine *rhs*, where the first dimension only contains five elements, and the second one has been selected as default. The speed-up achieved with this distribution is listed in the fourth column. The fifth column shows the simulated execution time when evaluating the solution suggested by our tool, and its respective speed-up can be seen in the last column. All times are expressed in miliseconds.

Name	Sequential time	Default parallel time	Speed-up	Suggested parallel time	Speed-up
eflux	1727.6	282.6	6.1	282.6	6.1
tred2	707.3	2473.0	0.3	546.8	1.3
rhs	4371.6	1218.0	3.6	1053.0	4.2
intba1	400.9	138.7	2.9	81.3	4.9
comp	3618.6	1624.7	2.2	653.5	5.5
erlebacher	2571.1	4798.1	0.5	1795.5	1.4

Table 4: Run-time behavior of the selected programs.

From this table one can see that the mapping selected by the tool for the first routine is the default one. The same was selected in [Gupt92]. For routine *tred2*, the mapping selected achieves a poor speed-up of 1.3, although the simulated execution time for the default one is more than 3 times the sequential execution time. In [Gupt92] the mapping selected was the default one, but in a cyclic manner. Our tool does not yet generate cyclic distributions. However, the simulated execution time for the cyclic distribution is very close to the default one. Routine *rhs* results in a speed-up of 4.2 with the suggested mapping, this is, distributing the third dimension of all arrays; while a speed-up of 3.6 is achieved with the default one. The parallel version of this routine given

in the xHPF benchmark set suggests a distribution of the fourth dimension of all arrays; the run-time behavior of these two alternatives is very similar. Routines *intba1* and *comp* also increase the simulated performance with respect to the default mapping, doubling its speed-up. The selected parallelization strategy is the same than the one suggested in the hand coded parallel version of these routines included in the xHPF benchmark set. Finally, program *erlebacher* reduces by 30% the sequential execution time when distributing arrays as suggested by the tool, while the simulated execution time of the program with the default distribution is almost twice the sequential one. The selected mapping strategy parallelizes almost all outer loops in the program. The reason for the poor speed-up is the amount of communication involved, in particular some reductions not detected by the parser module of our tool.

6 Related Work

Most of the fully automatic data distribution methods [LiCh90, Gupt92, Whol92, CGST93] perform the job in two main steps: alignment and distribution. The main differences between different methods, is the kind of structure selected to represent the problem, and the way used to formulate and solve it.

For the alignment step, [LiCh90] proposal defines and uses the Component Affinity Graph (CAG) to represent alignment preferences, and it uses a heuristic algorithm to solve it. [Gupt92] proposal uses the CAG, but weighted with data movement costs. To do this, a default distribution has to be assumed. This proposal is more accurate than the previous one when weighting the edges of the CAG, but the solver is based in the same heuristic algorithm. [Whol92] proposal uses the preference graph defined by [KnLS90] in the framework of SIMD machines. This graph includes alignment preferences to preserve parallelism, but the resolution when the graph is in conflict is also based in heuristics. [CGST93] defines the Alignment Distribution Graph (ADG) where nodes represent computations, and edges represent the number of data items being communicated along that edge. The alignment is found using a compact dynamic algorithm as a heuristic to determine the minimum cost. In all cases, the alignment is solved independently of the distribution, and the method used to find the solution is a heuristic algorithm.

Once the alignment has been decided, a distribution must be selected. [LiCh90] generates the communication routines involved in the program in a parametrized general form. Then, assuming a specific target machine, more accurate data movement cost functions can be obtained. Finally, this cost function is minimized and a particular distribution strategy is selected. [Gupt92] decides what dimensions to distribute (maximum two dimensions), assuming a default number of processors in each one and minimizing the total communication plus computation time. Then, if more than one dimension must be distributed, it decides what number of processors to assign to

each dimension generating all possible combinations. [Whol92] uses a hill climbing search method. Initially assigns all array elements to one processor and estimates this cost. Then it doubles the number of processors and chooses the dimension to assign the new ones, until all available processors are utilized or until distribution no longer reduces the total cost. This method considers the case in which data is distributed through less processors than available. [CGSS94] selects a set of candidate distributions based on the characteristics of the array objects, but not on any parallelism consideration. Then, using the divide-and-conquer paradigm, a dynamic distribution is determined. In all cases, the distribution is selected after the alignment has been fixed, and except for [LiCh90], an iterative method is used.

In our approach, the CPG is built, a graph with information about both the data movement and parallelism inherent in a block of code. The alignment and the distribution problems can then be solved in an unified way, and in a single step. This allows us to minimize communication while maximizing parallelism. The solution found is optimal, as it uses a general purpose linear 0-1 integer programming solver. In the framekork of automatic data layout, the use of linear 0-1 integer programming solvers was proposed by [BiKK94] in the frame of dynamic data layout.

There are some works done that consider dynamic layout, like [BiKK94, CrPe93], or mobile alignment like [ChGS93], but this is out of the scope of this paper. Some other approaches try to find a communication free data layout [RaSa91, HuSa91, ChSh93]. The alignment and distribution problems are solved together by using linear algebra and the solution, when found, is optimal. But most scientific code is not communication free, so these methods can not find a valid solution.

7 Conclusions and Future Work

In this paper we have presented a novel approach to perform automatic data distribution for distributed memory architectures. The novelty of the approach resides in the following two aspects: problem formulation and problem solving. Most of the current approaches to perform automatic data distribution work in two steps: first they solve the alignment problem (how arrays are aligned each other) and then they solve the distribution problem (which array dimensions are distributed and how many processors are allocated to each of them). In this paper we use the Communication-Parallelism Graph (CPG), a data structure able to hold all the information required to solve the two problems altogether. The CPG contains edges that show communication constraints and edges that show parallelization constraints in the program. Edges are weighted according to their cost, in terms of data movement and computation time. Constraints in the graph are formulated using a linear 0-1 integer programming model, where the problem to solve is to find a path in the CPG that minimizes the overall execution time of the application. A general purpose linear programming solver has been used in our experiments.

The preliminary experimental results show the quality of the data distributions generated by the tool, and the feasibility of using linear 0-1 integer programming models to solve these kind of problems. In addition, it allows us to find an optimal solution in a small amount of time. We have compared the solution generated by the solver with a default solution that would be generated by a naive tool.

We have shown all the details about how the problem can be solved for one-dimensional array distributions. The main ideas about multi-dimensional array distributions have been outlined. A lot of additional aspects should be considered in the problem formulation in order to improve the accuracy of the model and therefore the quality of the solutions generated, such as integrating communication optimizations [HiKT92] (detection and elimination of redundant communication, overlapping of computation and communication, or combination of communication messages) or control flow analysis.

We are also working in the direction of extending this approach to dynamic data mappings, where a set of computational intensive phases are detected, the mapping for each of them is obtained, and remapping actions (realignment and redistribution) between phases introduced.

8 Acknowledgements

We would like to thank the members of the DDT eam - Mercè Gironès, and M. Luz Grande - for their support for the implementation of the tool, and to Uli Kremer for the fruitful discussions in the topic. The authors especially thank Elena Fernández for her insight comments and suggestions in the specification of the linear programming model. This work has been supported by the Ministry of Education of Spain under contract TIC880/92, the CEPBA (European Center for Parallelism of Barcelona), CONVEX Computer Corporation, and the CIRIT under grant BE94/1-100.

9 References

[AGGL94] E. Ayguadé, J. Garcia, M. Gironès, J. Labarta, J. Torres and M. Valero, "Detecting and Using Affinity in an Automatic Data Distribution T ool". Proc. of the 7th Annual W orkshop on Languages and Compilers for Parallel Computing, August 1994.

[APR94] Applied Parallel Research Inc., "xHPF Version 1.2, User's Guide". May 1994 Release.

[BiKK94] R. Bixby, K. Kennedy and U. Kremer, "Automatic Data Layout Using 0-1 Integer Programming". Proc. of the International Conference on Parallel Architectures and Compilation Techniques, August 1994.

[CGST93] S. Chaterjee, J. R. Gilbert, R. Schreiber and S.-H. Teng, "Automatic Array Alignment in Data-Parallel Programs". Proc. of the 20th Annual ACM Symposium on Principles of Programming Languages, January 1993.

[CGSS94] S. Chaterjee, J. R. Gilbert, R. Schreiber and T. J. Sheffler, "Array Distribution in Data-Parallel Programs". Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computing, August 1994.

[ChGS93] S. Chaterjee, J. R. Gilbert and R. Schreiber, “Mobile and Replicated Alignment of Arrays in Data-Parallel Programs”. Proc. of Supercomputing’93, November 1993.

[ChSh93] T.-S. Chen and J.-P. Sheu, “Communication-Free Data allocation Techniques for Parallelizing Compilers on Multicomputers”. 1993 Int. Conference on Parallel Processing, vol. II, August 1993.

[CrPe93] P. Crooks and R. H. Perrott, “An Automatic Data Distribution Generator for Distributed Memory MIMD Machines”. 4th International Workshop on Compilers for Parallel Computers, December 1993.

[FHKK90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng and M. Wu, “Fortran D Language Specification”. Technical Report CRPC TR 90-141, Department of Computer Science, Rice University, December 1990.

[GMSS93] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang and K. Burke, “PTRAN II - A Compiler for High Performance Fortran”. 4th International Workshop on Compilers for Parallel Computers, December 1993.

[Gupt92] M. Gupta, “Automatic Data Partitioning on Distributed Memory Multicomputers”. PhD thesis, University of Illinois at Urbana-Champaign, September 1992. Also available as technical report UILU-ENG-92-2237 and CRHC-92-19.

[HiKT92] S. Hiranandani, K. Kennedy and C.-W. Tseng, “Compiling Fortran D for MIMD Distributed-Memory Machines”. Communications of the ACM, vol. 35, August 1992.

[HPFF93] High Performance Fortran Forum, “High Performance Fortran Language Specification. Version 1.0”. Scientific Programming, May 1993.

[HuSa91] C.-H. Huang and P. Sadayappan, “Communication-Free Hyperplane Partitioning of Nested Loops”. Proc. of the 4th International Workshop on Languages and Compilers for Parallel Computing, August 1991.

[KeKT90] K. Kennedy, K. McKinley and C.-W. Tseng, “Interactive Parallel Programming Using the ParaScope Editor”. Technical Report CRPC-TR90096, Center for Research on Parallel Computation, Rice University, October 1990.

[KnLS90] K. Knobe, J. D. Lukas and G. L. Steele, Jr., “DataOptimization: Allocation of Arrays to Reduce Communication on SIMD Machines”. Journal of Parallel and Distributed Computing, vol. 8, February 1990.

[Krem94] U. Kremer, “NP--completeness of dynamic remapping”. 4th International Workshop on Compilers for Parallel Computers, December 1993.

[LiCh90] J. Li and M. Chen, “Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays”. Proc. of the 3rd Symposium on the Frontiers of Massively Parallel Computationd, October 1990.

[LiCh91] J. Li and M. Chen, “Compiling Communication-efficient Programs for Massively Parallel Machines”, IEEE Trans. on Parallel and Distributed Systems, vol. 2, no. 3, July 1991.

[LIND94] LINDO Systems Inc., “LINGO Optimization Modeling Language”. © April 1994 by LINDO Systems Inc.

[RaSa91] J. Ramanujam and P. Sadayappan, “Compile-Time Techniques for Data Distribution in Distributed Memory Machines”. IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 4, October 1991.

[Tsen93] C. Tseng, “An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines”. PhD thesis, Rice University, January 1993. Rice COMP TR93-199.

[Whol92] S. Wholey, “Automatic Data Mapping for Distributed-Memory Parallel Computers”. Proc. of the 1992 ACM International Conference on Supercomputing, July 1992.

[ZBCM91] H. Zima, P. Brezany, B. Chapman, P. Mehrotra and Schwald, “Vienna Fortran - A Language Specification”. Technical Report, Austrian Center for Parallel Computation, University of Vienna, 1991.

Appendix

In the implementation, nodes source and sink are not considered. An integer identifier n has been assigned to each non empty set of edges. In the example there are four different sets of edges: Y_{AB}, Y_{BC}, Y_{CD} and Y_{DD}, respectivelly. The set of all 0-1 integer variables is represented by Y, and it is declared as an array of dimensionality [n x d x d], where n is the number of patterns between different pairs of arrays, and d is the maximal dimensionality of all arrays. C is the array of the same dimensionality that holds the weights for each edge. Z represents the set of hyperedges and is declared as a vector of dimensionality [m], where m is the number of parallel loops. Vector T holds the corresponding weights associated to hyperedges.

```
! Constraints file for example routine;
! 1-D DISTRIBUTION. EDGE BASED MODEL;
MODEL:

SETS:
     ! Declaration of variables;
     maxdim /1..3/ : ;
     numedg /1..4/ : ;
     numhyp /1..3/ : ;
     edges (numedg, maxdim, maxdim) : Y, C;
     hyper (numhyp) : Z, T;
ENDSETS

DATA:
     ! Communication cost values;
     C =  ...list of (numedg x maxdim x maxdim) constant values...

     ! Parallelization cost values;
     T =  ...list of (numhyp) constant values...
ENDDATA

! Declaring all variables as 0-1 integer;
@FOR (edges: @BIN (Y); );
@FOR (hyper: @BIN (Z); );

! 1st constraint: the solution is a path, all edges are connected;
@FOR (maxdim(i):
     @SUM (maxdim(j): Y(1, j, i) ) = @SUM (maxdim(j): Y(2, i, j))
);
@FOR (maxdim(i):
     @SUM (maxdim(j): Y(2, j, i) ) = @SUM (maxdim(j): Y(3, i, j))
);
@FOR (maxdim(i):
     @SUM (maxdim(j): Y(3, j, i) ) = @SUM (maxdim(j): Y(4, i, j))
);

! 2nd and 3rd constraints: exactly one edge must be selected;
@FOR (numedg(n):
     @SUM (maxdim(i):
          @SUM (maxdim(j): Y(n, i, j) ) ) = 1
);

! 4th constraint: correct behavior of hyperedges;
@SUM (maxdim(j): Y(1, 1, j) ) >= Z(1);
@SUM (maxdim(j): Y(2, 3, j) ) >= Z(1);
@SUM (maxdim(i): Y(3, i, 1) ) >= Z(1);

@SUM (maxdim(j): Y(1, 2, j) ) >= Z(2);
@SUM (maxdim(j): Y(2, 2, j) ) >= Z(2);
@SUM (maxdim(i): Y(3, i, 2) ) >= Z(2);

@SUM (maxdim(j): Y(1, 1, j) ) >= Z(3);
@SUM (maxdim(j): Y(1, 3, j) ) >= Z(3);

! Objective function to minimize;
CostOfEdges = @SUM (edges: Y * C);
CostOfHyper = @SUM (hyper: Z * T);
Cost = CostOfEdges - CostOfHyper;

END
```

Automatic Data Layout For High Performance Fortran

Ulrich Kremer and Ken Kennedy, Rice University

The goal of High Performance Fortran (HPF) is to provide a simple yet efficient machine-independent parallel programming model. Besides the algorithm selection, the data layout choice is the key intellectual challenge in writing an efficient program in such languages. The performance of a data layout depends on the target compilation system, the target machine, the problem size, and the number of available processors. This makes the choice of a good layout extremely difficult for most HPF users. Allowing remapping of arrays at specific points in the program makes the selection of an efficient data layout even harder. If HPF is to find general acceptance, the need for data layout selection support has to be addressed. We believe that a tool that generates data layout specifications automatically is the appropriate way to provide the needed support. Based on the automatically generated data layouts, a user can further improve the data layout selection until the overall program performance reaches a level that is acceptable to him or her.

Our framework for automatic data layout selection builds and examines explicit search spaces of candidate data layouts. A candidate layout is an efficient layout for some part of the program. After the generation of search spaces, a single candidate layout is selected for each program part, resulting in a data layout for the entire program. A good overall data layout may require the remapping of arrays between program parts. A performance estimator based on a compiler model, an execution model, and a machine model is used to predict the execution time of each candidate layout and the costs of possible remappings between candidate data layouts. The machine model uses training sets to determine the costs of arithmetic operations and simple communication patterns.

In the framework, instances of NP-complete problems are solved during the construction of candidate layout search spaces and the final selection of candidate layouts from each search space. Rather than resorting to heuristics prematurely, the framework capitalizes on state-of-the-art 0-1 integer programming technology to compute optimal solutions of these NP-complete problems.

The framework has been implemented as part of a data layout assistant tool. Preliminary experimental results based on the Fortran D compiler as the target HPF compiler, and the iPSC/860 as the target machine architecture indicate that the framework is efficient and generates good data layouts.

Automatic Data Layout for High Performance Fortran

Ken Kennedy Ulrich Kremer

CRPC-TR94498-S
December, 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
Houston, TX 77005-1892

Revised April 1995

Automatic Data Layout for High Performance Fortran*

Ken Kennedy Ulrich Kremer†

Department of Computer Science
Rice University

Abstract

High Performance Fortran (HPF) is rapidly gaining acceptance as a language for parallel programming. The goal of HPF is to provide a simple yet efficient machine independent parallel programming model. Besides the algorithm selection, the data layout choice is the key intellectual step in writing an efficient HPF program. The developers of HPF did not believe that data layouts can be determined automatically in all cases. Therefore HPF requires the user to specify the data layout. It is the task of the HPF compiler to generate efficient code for the user supplied data layout.

Since many compiler decisions are driven by the specified data layout, predicting the performance of a data layout is extremely difficult. The choice of a good data layout depends on the HPF compiler used, the target architecture, the problem size, and the number of available processors. Allowing remapping of arrays at specific points in the program makes the selection of an efficient data layout even harder.

Although finding an efficient data layout fully automatically may not be possible in all cases, HPF users will need support during the data layout selection process. In particular, this support is necessary if the user is not familiar with the characteristics of the target HPF compiler and target architecture, or even with HPF itself. Therefore, tools for automatic data layout and performance estimation will be crucial if the HPF is to find general acceptance in the scientific community.

This paper discusses a framework for automatic data layout for use in a data layout assistant tool for HPF. Since the data layout assistant is not embedded in a compiler and

*This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. This work was also sponsored by ARPA under contract #DABT63-92-C-0038. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

†**Corresponding author**; e-mail: kremer@cs.rice.edu; phone: (713) 527-6077; address: Department of Computer Science, Rice University, 6100 S. Main, Houston, Texas 77005–1892

will run only a few times during the tuning process of an application program, the framework can use techniques that may be too computationally expensive to be included in a compiler. The framework is parameterized with respect to the HPF compiler, the machine architecture, the problem size, and the number of available processors. The automatically selected layout may contain remappings if they are found profitable.

The framework has been implemented as part of a data layout assistant tool in the D system. The paper reports preliminary experimental results based on the Fortran D compiler as the target HPF compiler, and the iPSC/860 as the target machine architecture. The experiments indicate that the framework is efficient and generates good data layouts.

1 Introduction

Data parallel languages such as High Performance Fortran (HPF) [KLS+94] require the user to specify the layout of an application's data across the processors of a parallel machine. Many complex considerations have to be taken into account in choosing an efficient data layout for an application. The choice is application, compiler, and machine dependent. In addition, an application may consist of a set of computations that each require distinct data mappings for best performance.

HPF compilers typically perform many optimizations based on the specified data layout. The actual optimizations performed may vary between compilers. Different parallel machine architectures have different communication and computation costs, resulting in distinct optimal balances between communication and computation for each architecture. All these factors make it extremely difficult for a user to predict the performance impact of a particular data layout for his or her application.

This paper discusses a framework for automatic data layout for regular problems in the context of a programming assistance tool such as the D system [ACG+94]. The application scenario for our proposed data layout assistant tool is shown in Figure 1. Typically, regular problems represent data objects as dense arrays as opposed to a sparse representation. Regular problems allow the compilation system to determine the communication requirements and to perform a variety of program optimizations at compile time. The framework assumes that different data layouts can be specified for different program sections. We envision the tool to be used to generate a first data layout for a sequential Fortran program without data layout statements, or to extend a partially specified data layout in a HPF program to a totally specified data layout.

Since the data layout assistant is not part of a compiler, our framework can use techniques that may be too computationally expensive to be included in a compiler. The data layout is optimized for a target compiler, a target architecture, a specific problem size, and the number of available processors. All these entities influence the performance of a data layout. This implies that these entities have to be known at tool invocation time. Note that an automatically generated data layout can be used for different problem sizes and numbers of processors, although its quality may be suboptimal.

The proposed framework for automatic data layout in the presence of dynamic data remapping consists of several steps. An overview of these steps is given in Section 2. A prototype implementation of our framework is described in Section 3. Experimental results

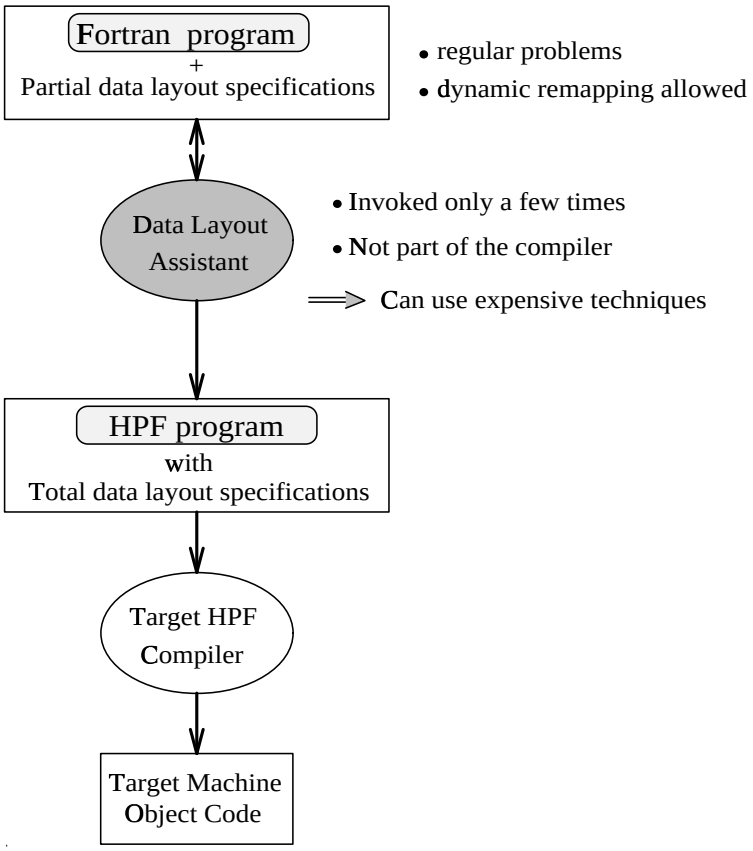


Figure 1: Automatic data layout as part of a programming environment

based on the prototype implementation are discussed in Section 4. The paper concludes with a discussion of related work, followed by a summary and discussion of future work in Section 5 and Section 6, respectively.

2 Framework for Automatic Data Layout

The framework for automatic data layout consists of four steps. In the first step the input program is partitioned into program segments. For each such program segment, the second step constructs a search space of promising candidate layouts. A candidate layout for a program segment is a mapping of every array referenced in the segment onto the target architecture. Heuristics are used to generate the candidate layout search spaces. In the third step each candidate layout is evaluated in terms of its estimated execution time. In addition, costs of possible remappings between candidate layouts are determined. The performance estimation is based on a compiler model, execution model, and machine model. Based on the estimated candidate layout costs and costs of possible remappings between candidate layouts, a single candidate layout from each search space has to be selected such that the overall cost is minimal. This selection process is performed in the fourth and last step of

our framework. In the remainder of this section each of the four steps is discussed in more detail.

2.1 Program Partitioning

The first step partitions the program into code segments, called program *phases*. In our current framework, data remapping is allowed only between phases. A *phase* is the outermost loop in a loop nest such that the loop defines an induction variable that occurs in a subscript expression of an array reference in the loop body. This operational definition does not allow the overlapping or nesting of phases. Other strategies for identifying program phases are a topic of current research. Transformations to improve phase recognition are beyond the scope of this paper.

The phase structure of the program is represented in the *phase control flow graph* (PCFG), an augmented control flow graph [ASU86] where each phase is represented by a single node. The graph is annotated with branch probabilities and loop control information. Branch probabilities can either be supplied by the user or are determined based on a guessing heuristic.

2.2 Layout Search Spaces Construction

The second step of the framework for automatic data layout constructs explicit search spaces of candidate data layouts for each phase. Promising candidate layouts for a phase are generated based on their expected performance as part of an efficient data layout for the entire program.

A data layout in HPF is defined in two stages, typically referred to as *alignment* and *distribution*. Arrays are aligned relative to each other by specifying a mapping of their elements to the same array of virtual processors, called a *template*. Every array element aligned with a template is mapped to a real processor by distributing the template onto the processors of the target architecture.

The use of explicit search spaces is an important design decision in our framework for automatic data layout. Explicit alignment search spaces allow the framework to postpone the evaluation of an alignment candidate until all distribution candidates are known. Therefore, promising alignment candidates are not eliminated prematurely, but are evaluated in combination with the selected candidate distributions. In addition, explicit search spaces provide a natural basis for user interaction. The user can browse through the search spaces and influence the selection process by adding or deleting candidate data layouts.

The current framework determines a single template for the entire program based on the maximal dimensionalities and maximal dimensional extents of the arrays in the program. All alignments and distributions are specified based on this *program template*. Corresponding to the two stage mapping, the framework first builds search spaces of promising candidate alignments for each phase. If arrays have fewer dimensions than the program template, alignment analysis may generate different embeddings for the arrays. Then, distribution analysis uses the alignment search spaces to build candidate data layout search spaces of reasonable alignments and distributions for each phase. Alignment and distribution analysis

is discussed in the next two sections.

2.2.1 Alignment Analysis

Alignment analysis takes the phase control flow graph as input and generates explicit alignment search spaces for each phase. Heuristics have to be used to determine a reasonably sized set of alignment candidates that will guarantee a good overall performance for most applications. This section discusses basic operations that are needed to identify and represent alignment preferences, to detect and resolve conflicting alignment preferences, and to compare candidate alignments. The comparison of alignment candidates is important in order to avoid redundant alignment information in the alignment search spaces.

The basic operations form the building blocks for implementing different heuristics and strategies for the alignment search space construction. The heuristic implemented in our prototype tool is discussed in Section 3.1. Other heuristics such as the one described by Lee and Tsai [LT93] can be easily implemented in our framework.

There are two types of alignment preferences, namely *inter–dimensional* and *intra– dimensional* alignment [LC90, KLS90a, CGST93]. The current framework does not perform intra-dimensional alignment analysis, i.e., assumes canonical offset and stride alignments. The discussion of the basic operations will be restricted to inter-dimensional alignment preferences.

Identification and Representation of Alignment Preferences

A central representation for the inter-dimensional alignment problem is the weighted, undirected *component affinity graph* (CAG) introduced by Li and Chen at Yale University [LC90]. It represents the alignment preferences of arrays that are coupled in a computation. A d-dimensional array is represented in the CAG by d nodes, one node for each dimension. Alignment preferences between dimensions of distinct arrays are represented as edges between the corresponding nodes. The weights of the edges reflect the relative importance of alignment preferences.

Detection of Alignment Conflicts

Assume that d is the dimensionality of the program template. A solution to the inter-dimensional alignment problem is a partitioning of the nodes in the CAG into d partitions such that no two nodes representing dimensions of the same array are in the same partition. A CAG contains a *conflict* if there is a path between two nodes that represent distinct dimensions of the same array. The test for alignment conflicts is linear in the size of the CAG, since it involves the solution of reachability problems between nodes in the CAG.

Alignment Conflict Resolution

A conflict implies that every solution of the corresponding inter-dimensional alignment problem will have alignment preferences that cannot be satisfied. A good solution tries to minimize the weights of the edges that cross partitions and therefore cannot be satisfied. Li and Chen showed that finding the optimal solution for the inter-dimensional alignment prob-

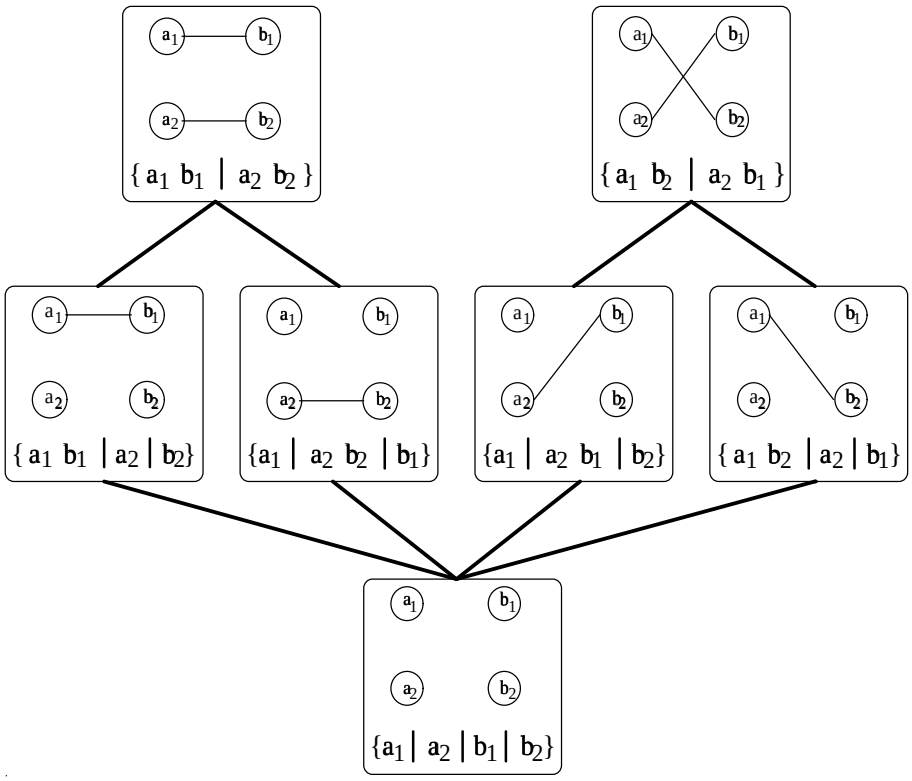


Figure 2: Inter-dimensional alignment information lattice for two arrays a and b. Both arrays have two dimensions. Each conflict-free CAG is shown with its corresponding node partitioning. The bottom element of the lattice is the CAG without edges, i.e., the partitioning $\{a_1 \mid a_2 \mid b_1 \mid b_2\}$.

lem is NP-complete [LC90]. Instead of using a heuristic, the current framework formulates the inter-dimensional alignment problem as an efficient 0–1 integer programming problem. A detailed description of our 0–1 formulation and a proof of its correctness is given in the appendix.

Comparison of Alignment Preferences

The inter-dimensional alignment information of a conflict-free CAG can be represented as a partitioning of its nodes. Each partition in the partitioning represents a connected component in the CAG. The set of all possible conflict-free, inter-dimensional alignments of a set of arrays forms a semi-lattice [Hec77]. The bottom element of the lattice is the CAG that contains no alignment information, i.e., the graph contains no edges and therefore its partitioning consists of partitions that contain only single nodes. Figure 2 shows an example lattice for two two-dimensional arrays.

The partial order $\sqsubseteq$ defined over the set of conflict-free CAGs is that of partitioning refinement. Assume that CAG_1 and CAG_2 are two conflict-free CAGs, then $CAG_1 \sqsubseteq CAG_2$ if and only if the node partitioning of CAG_1 is a refinement of the node partitioning of CAG_2. A partitioning X is a refinement of a partitioning Y, if for each partition $x \in X$ there is a

partition $y \in Y$, such that $x \subseteq y$. Assuming that partitions are implemented using hashing, and the elements in X and Y are tagged with their partition membership, then the test "X is refinement of Y" is linear in practice in the number of elements in all partitions of X. In other words, the complexity of computing $CAG_1 \sqsubseteq CAG_2$ is linear in practice in the number of nodes in CAG_1.

2.2.2 Distribution Analysis

Distribution analysis is performed after alignment analysis. A candidate distribution can map single template dimensions either by `block`, `cyclic`, or `block-cyclic` onto the target architecture, or replicate dimensions on each processor. In addition, a candidate distribution specifies the number of processors in each distributed dimension. Different heuristics can be used to determine a suitable set of promising distribution candidates. Once the distribution candidates have been determined, the cross product of alignment candidates and distribution candidates defines the candidate data layout search spaces for each phase.

2.3 Performance Estimation

After the generation of the search spaces each candidate data layout is evaluated in terms of its expected execution time for its phase. In addition, execution time estimates are needed for possible remappings between candidate data layouts.

The performance estimator uses a compiler model to determine where and what kind of communication will be generated for a given candidate data layout and its phase. The compiler model is parameterized with respect to the transformations and communication optimizations that may be performed by the target compiler. Note that the compilation process needs to be simulated for performance purposes only. Special cases that have a small impact on the the overall performance, but must be handled by a real compiler in order to generate correct code, may be ignored in the performance estimation compiler model. For instance, the compiler model implemented as part of our prototype ignores code that is generated for boundary processors in loops. A more detailed description of the implemented compiler model can be found in Section 3.

Once locations and types of compiler generated communications are known for a candidate layout and its phase, an execution model is used to estimate the performance effects of synchronizations induced by the communications. Communication inside a phase may lead to a pipelined execution of the loop. In contrast, communication outside of the phase may result in a loosely synchronous execution scheme.

The actual costs of communication operations and basic computations for the target machine architecture can be determined based on the training set approach [BFKK91]. Training sets for global communication operations can be used to estimate the cost of remappings between candidate data layouts.

2.4 Layout Selection

As the result of the performance estimation step, performance numbers in terms of relative execution times are available for all candidate data layouts and possible remappings between

layouts. In the last step of our framework for automatic data layout, a single candidate layout has to be selected from each search space of each phase such that the resulting set of candidate layouts has minimal overall cost. The overall cost is determined by the costs of each selected candidate layout and the required remapping costs between selected layouts. Note that the optimal data layout for a program may consist of candidate data layouts that are each suboptimal for their phases.

In our framework, the final layout selection problem is formulated as a graph problem. The *data layout graph* has one node for each candidate layout. Edges represent possible remappings between layouts. Nodes and edges are weighted by their relative execution times. The optimal selection problem has been shown to be NP-complete [Kre93]. We discuss a translation of the graph problem into a 0–1 integer programming formulation elsewhere [BKK94]. We found our 0–1 formulation to be efficient in practice.

3 Prototype Implementation

A prototype of our data layout assistant tool has been implemented as part of the D system [ACG+94]. The current target HPF compiler is the Fortran D prototype compiler [Tse93]. Fortran D [FHK+90] shares many features with HPF since it was one of the main contributors to the HPF language proposal. The current target architecture is Intel's iPSC/860. The prototype tool does not perform inter-procedural analysis. Non-linear control flow in input Fortran programs is restricted to `Do` loops and `If` statements.

The implemented alignment analysis is discussed in Section 3.1. Distribution analysis generates only one-dimensional, `block` distributions. This restriction is due to the fact that the compiler model implementation mimics the program analysis steps in the Fortran D prototype compiler which does not support multi-dimensional distributions.

The experiments discussed in Section 4 assume a target compiler that performs message coalescing and message vectorization, but does not perform coarse grain pipelining, loop interchange, or loop distribution. The compiler model simulates this target compiler. The execution model uses data dependence information to detect processor synchronization. Phases are classified as either pipelined, loosely synchronous, or reductions. Performance estimates for basic computations and communication patterns are based on machine level training sets for the iPSC/860 [BFKK91].

The current implementation uses 79 training sets that measure basic computations such as `real` and `double` floating point operations, and basic communication patterns such as nearest neighbor communication, single send/receive pairs, broadcasts, reductions, and transpose operations. For each communication pattern there are training sets for different numbers of processors, different memory access patterns, and different observable message latencies. The current implementation distinguishes between a unit or non-unit stride memory access pattern, and high or low latency messages. A non-unit memory access pattern usually requires message buffering. Low latency message costs are used to estimate the communication costs in pipelined phases where computation and communication can be overlapped. In contrast, message costs for loosely synchronous phases are based on high latency training sets.

The prototype tool solves NP-complete problems during alignment analysis and the final data layout selection step. Instances of these problems are translated into 0–1 integer

programming problems suitable to be solved by *CPLEX*[1], a linear integer programming tool and library, partly developed by Robert Bixby at Rice University [Bix92]. The prototype tool builds the required constraint matrices internally, and directly calls the CPLEX routines without creating any intermediate files.

3.1 Heuristic for Construction of Alignment Search Spaces

The alignment search spaces are initialized with the CAGs of their phases. If a CAG has inter-dimensional alignment conflicts, the conflict is resolved and the resulting CAG is used for the initialization. After initialization, the phases are partitioned into classes or *partitions* such that their merged CAGs are conflict-free. Each class of phases is represented by its merged, conflict-free CAG. The current prototype uses a greedy partitioning algorithm that visits the phases, i.e., the nodes in the PCFG in reverse postorder (rPOSTORDER) [Hec77], and merges their CAGs as long as no conflict is detected. Once a conflict is encountered, a new partition is created and initialized with the single phase that led to the conflict. The partitioning algorithm terminates after all phases have been visited.

The main step of the heuristic consists of exchanging alignment information between different phase partitions. An *imported* alignment candidate is an alignment scheme from another partition. The import process merges the CAG of the sink partition of the import operation and the CAG of the source partition of the import operation, after increasing the edges weights of the source CAG by some constant factor. The edge increase guarantees that the alignment preferences of the source CAG will dominate the alignment preferences of the sink CAG. This is important if the merged CAG has an alignment conflict. The imported alignment candidate is the alignment scheme resulting from solving possible conflicts in the merged CAG, and restricting the resulting alignment information to those arrays that are referenced in the sink partition of the import operation.

The current prototype imports each optimal alignment candidate of another partition. If the phase partitioning has p partitions, then each final partition alignment search space can have at most p candidates. In order to avoid duplication of alignment information, the imported alignment candidate is only inserted into the search space if its alignment information is not weaker or equal to any alignment information already in the search space.

Finally, candidate alignment schemes for phase partitions are translated into candidate alignments for each phase in the partition. The final representation of the alignment information of a conflict-free CAG is a partitioning of its nodes into d classes, where d is the dimensionality of the program template. The current prototype uses a canonical collapsing order, if the conflict-free CAG has more than d connected components.

4 Experimental Results

The target compiler for the experiments described in this section was the Fortran D compiler prototype [Tse93] with loop interchange and coarse-grain pipelining disabled. The target architecture was Intel's iPSC/860.

[1] *CPLEX* is a trademark of CPLEX Optimization, Inc.

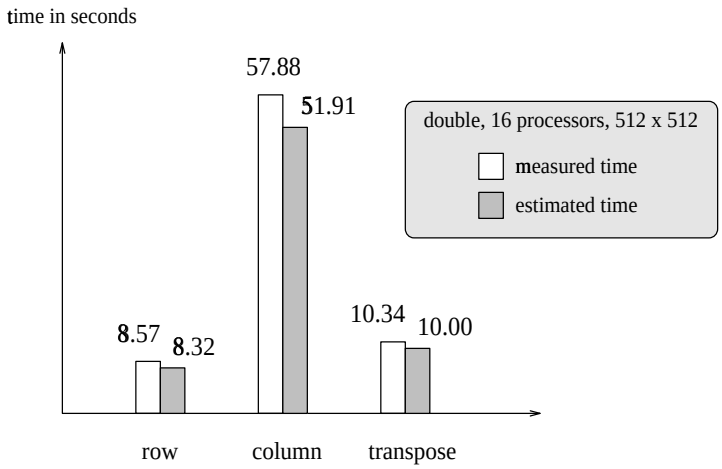


Figure 3: Example test case for ADI with three possible data layouts

It is important to note that it is not the goal of the experiments to evaluate the quality of any target compiler, but to show the ability of the data layout assistant tool to simulate the target compiler and to correctly estimate the relative performance of the candidate layouts in its generated search spaces. The current generation of commercially available HPF compilers may not perform advanced transformations such as coarse-grain pipelining. The quality of a data layout for a program is always relative to the HPF compiler that is used to compile the program.

The experiments were based on three programs, an alternating direction implicit integration kernel (Adi), a 3D tridiagonal solver based on ADI integration and developed by Thomas Eidson at ICASE (Erlebacher), and a grid generation program, adapted from the SPEC benchmark suite by Applied Parallel Research (Tomcatv). Parts of the programs were rewritten into a data-parallel programming style that allows better compile time analysis [KR94].

The automatic data layout tool was applied to each program for different test cases. A test case consists of a data type for the arrays in the program, a problem size, and a number of processors used. Figure 3 shows a single test case for the Adi kernel and its results. The test case is for double precision arrays, 16 processors, and a problem size of 512×512. For each test case, the overall execution times of promising data layouts for the entire program were measured and compared to execution times predicted by the prototype data layout assistant tool. For all three programs, the prototype tool did not miss any promising data layouts. The remaining questions are whether the tool ranked the data layout alternatives correctly, and whether the best predicted data layout alternative was also the best measured alternative. For the Adi kernel test case shown in Figure 3 the prototype tool picked the best data layout, namely a static row-wise data layout, and also ranked the data layout alternatives correctly.

To perform the comparison, each program was compiled with the target compiler under its set of promising data layouts. When necessary, the output of the Fortran D compiler was modified by hand to ensure correct code. The resulting SPMD node programs were executed

and timed on the target architecture. In the remainder of this section, we will discuss the results for each program in more detail.

Erlebacher: We used an inlined version of Erlebacher for the experiments, since the prototype implementation of the data layout assistant does not perform inter-procedural analysis. Erlebacher has 40 phases. There are no inter-dimensional alignment conflicts. The data layout selection step generated a 0–1 problem with 327 variables and 190 constraints. CPLEX solved the problem in 120 milliseconds on average on a SPARC-10.

The program consists of a three symmetric computations, each along one of the dimensions of the problem. The computations share access to a 3-dimensional, read-only array. All four 3-dimensional arrays are aligned canonically, i.e., there is no inter-dimensional alignment conflict. The choice of a static data layout leads to cross-processor dependences in exactly one of the three symmetric computations. Since the target compiler performs message vectorization but no coarse-grain pipelining or loop interchange, the particular loop order in the partitioned loops will determine the granularity of the resulting pipelined execution. For instance, if the third dimension is distributed, and the induction variable in the third dimension carries a dependence, a message will be generated at the loop level that carries the dependence. If the induction variable is defined at the outermost loop level, the generation of a message at that level will lead to a sequential execution of the loop. On the other hand, if the induction variable is defined at the innermost level, the execution of the loop will be fine-grain pipelined. Since the loop order is the same in each of the symmetric computations, the distribution of the second dimension will lead to a medium-grain pipeline. The granularity of this pipeline will depend on the problem size and the data type.

We measured 22 test cases. Distributing the first dimension resulted in introducing a fine-grain pipeline which was never profitable. Introducing a coarse-grain pipeline by distributing the second dimension was the best choice in 9 cases. The last possible static data layout, namely distributing the third dimension, resulted in the sequential execution of one of the three symmetric computations. This choice was the best in 2 cases. Finally, using a dynamic data layout by remapping the read-only array once between a pair of symmetric computations was the best choice in 11 cases.

The prototype tool determined the best layout in 11 cases. Since the performance of the dynamic data layout and the static layout that distributes the second dimension were very close, the tool failed to rank them correctly in some cases. However, the incorrect ranking would have resulted in a performance loss of at most 12.7% as compared to the best possible data layout choice.

Adi: In contrast to the computation in Erlebacher, Adi solves a two-dimensional problem. The program has 9 phases. There are no inter-dimensional alignment conflicts. The solution of the 0–1 data layout selection problem took CPLEX 60 milliseconds on average on a SPARC-10. The problem had 61 variables and 53 constraints.

We measured 40 test cases, one of which is shown in Figure 3. Distributing the second dimension (column layout) resulted in the sequential execution of two phases. This was always the worst choice. Distributing the first dimension (row layout) introduced a fine-grain pipeline in two phases and resulted in the best possible data layout in 24 cases. In the remaining 16 cases, a dynamic layout that remaps the arrays between row and column

sweeps (transpose layout) was the best data layout choice.

The prototype tool selected the best data layout in 37 cases. In all these cases the relative rankings of data layout alternatives were correct. In 3 cases the automatically chosen layout was suboptimal. The worst case performance loss due to the suboptimal selection was 9.3% as compared to the best possible choice.

Tomcatv: In contrast to Erlebacher and Adi, Tomcatv has inter-dimensional alignment conflicts for two of its 2-dimensional arrays. The assistant tool partitioned the 16 phases into two classes and exchanged their inter-dimensional alignment information. The resulting alignment search spaces for each phase had two entries. Together with the two possible single dimension distributions, the final data layout search space contained four candidate layouts for most phases. Some phases had search spaces with only two entries, since the projection of phase partition layouts onto single phase layouts resulted in identical candidate data layouts.

The two inter-dimensional alignment conflicts were translated into 0–1 problems with 312 variables and 530 constraints. Although the sizes of the two problems are the same, their objective functions are different, since the edge weights in the two merged CAGs are not identical. The sizes of the 0–1 problems are quite large since the program contains many scalar temporaries that are represented in the CAG as if they had been scalar expanded. On a SPARC-10, CPLEX solved the two problems in 480 and 1030 milliseconds on average. The 0–1 formulation of the data layout selection problem had 248 variables and 146 constraints. CPLEX determined the optimal solution in 140 milliseconds on average on a SPARC-10.

We measured 12 test cases. In all cases distributing the second dimension was the best choice. A dynamic data layout was never profitable. In all cases the prototype tool selected the best data layout.

5 Related Work

The problem of automatic data layout has been addressed by many researchers [AL93, CHZ91, CGS93, CGST93, Gup92, HA90, Keß93, KLS90b, KLD92, LT93, LC91, RS89, Who91]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data layouts, the compilation system, and the target machine architecture.

Our work is similar in nature to the recent work done by Anderson and Lam at Stanford University [AL93], Chatterjee, Gilbert, Schreiber, and Sheffler at RIACS and Xerox Parc [CGSS94], Ayguadé, Garcia, Girones, Labarta, Torres and Valero at the University of Catalunya in Barcelona, [AGG+94], and Ning, Van Dongen, and Gao at CRIM and McGill University [NDG95].

In contrast to previous work, our framework is designed to be used inside a data layout assistance tool and not inside a compiler. To support user interaction, the framework builds and examines explicit search spaces of possible candidate layouts. Since the tool is outside of the target compiler, a target compiler performance model is a crucial component of our framework. In addition, the framework can use techniques that may be too expensive to be included in a compiler.

6 Summary and Future Work

The paper discussed a new framework for automatic data layout designed to be used in a data layout assistant tool. Since the tool is not part of a compiler, automatic data layout techniques are used that may be too expensive to be included in a compiler. The framework has been implemented in a prototype data layout assistant tool.

The prototype implementation uses the latest and most powerful general purpose techniques for linear and integer programming to solve two NP-complete problems optimally. Experiments based on three scientific programs and program kernels indicate that our framework is efficient and generates good data layouts. All encountered instances of NP-complete problems were solved in less than 1.1 seconds. For all three programs, the generated data layouts were optimal or close to optimal within 12.7%.

We are currently working on a compiler model that will allow multi-dimensional distributions. We will perform more experiments to further verify the efficiency of our data layout framework, and to verify the quality of the generated data layouts. The presented framework for automatic data layout will be extended to handle programs that consist of multiple procedures.

References

[ACG+94] Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Chau-Wen Tseng, and Scott Warren. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, 1994.

[AGG+94] E. Ayguadé, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[AL93] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[Bix92] R. Bixby. Implementing the Simplex method: The initial basis. *ORSA Journal on Computing*, 4(3), 1992.

[BKK94] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0–1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, August 1994.

[CGS93] S. Chatterjee, J.R. Gilbert, and R. Schreiber. The alignment-distribution graph. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[CGSS94] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[CGST93] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.

[CHZ91] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, September 1992.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, NY, 1977.

[Keß93] C.W. Keßler. Knowledge–based automatic parallelization by pattern recognition. In C.W. Keßler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 110–135. Verlag Vieweg, Wiesbaden, Germany, 1993.

[KLD92] K. Knobe, J.D. Lukas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, 1992.

[KLS90a] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[KLS90b] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[KR94] U. Kremer and M. Ramé. Compositional oil reservoir simulation in Fortran D: A feasibility study on Intel iPSC/860. *International Journal of Supercomputer Applications*,

8(2):119–128, Summer 1994. Also available as Technical Report CRPC-TR93335, Center for Research on Parallel Computation, Rice University.

[Kre93] U. Kremer. NP-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as technical report CRPC-TR93-330-S (D Newsletter #8), Rice University.

[LC90] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(4):213–221, August 1991.

[LT93] P. Lee and T-B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[NDG95] Q. Ning, V. V. Dongen, and G. R. Gao. Automatic data and computation decomposition for distributed memory machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1995.

[RS89] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, Houston, TX, January 1993. Rice COMP TR93-199.

[Who91] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

Appendix

Alignment Conflict Resolution as a 0–1 Problem

This section describes the translation of an instance of the inter-dimensional alignment problem into an instance of a 0–1 integer programming problem with linear constraints. Note that there are many possible translations. Experiments showed that our formulation is very promising. A proof of the correctness of our formulation is also provided in this appendix.

Definition An instance of the inter-dimensional alignment problem with d dimensions consists of finding a d-partitioning of an undirected, weighted component affinity graph (CAG) such that the sum of the weights of the edges that cross distinct partitions is minimized.

Definition An instance of the 0–1 problem consists of a set of variables X, a set of linear constraints over the variables in X, and a linear objective function with domain X. A solution to an instance of the 0–1 problem is a function $s_{01} : X \rightarrow \{0, 1\}$ that minimizes (or maximizes) the objective function while respecting the constraints.

In the following, we will discuss the translation of a d-dimensional alignment problem into a 0–1 problem. We will assume that all arrays represented in the CAG have d or less dimensions. Let a_i be the node in the CAG that represents the i-th dimension of array a, $1 \leq i \leq dim(a)$, where $dim(a)$ denotes the number of dimensions of array a. Each such node is represented by d variables or switches in X, $a_{ik} \in X$, $1 \leq k \leq d$. The switch a_{ik} will be *on* if and only if the node a_i belongs to the k-th partition in the final solution. Let $e = (a_i, b_j)$ be an edge in the CAG. Each edge e is represented by d variables or switches in X, $a\$b^{ik}_{jk} \in X$, $1 \leq k \leq d$. In the final solution, the switch $a\$b^{ik}_{jk}$ is *on*, if and only if the sink and the source of the edge e belong to the same partition.

There are two types of constraints. *Node constraints* ensure that any solution is a d-partitioning of the CAG, and *edge constraints* identify edges with source and sink nodes in the same partition.

Node constraints: To ensure that an array dimension is in exactly one partition, constraints of the form $\sum_{k=1}^{d} a_{ik} = 1 \quad (type1)$ are introduced for each node a_i. Two dimensions of the same array must not be in the same partition. This property is enforced by the following constraints, one constraint for each pair of an array a and a partition k, $1 \leq k \leq d$: $\sum_{i=1}^{dim(a)} a_{ik} \leq 1 \quad (type2)$. Note that in the case of an embedding of array a, $dim(a) < d$, some partition k will not contain any CAG node associated with a. There are $|N|$ node constraints of $(type1)$ and $d|Arrays|$ node constraints of $(type2)$, where N is the set of nodes in the CAG, and $Arrays$ is the set of arrays represented in the CAG.

Edge constraints: The formulation of the inter-dimensional alignment problem uses counting arguments on the number of incoming and outgoing edges of nodes in the CAG. For each node a_i, constraints are introduced for incoming and outgoing edges. The translation requires a directed graph. The particular direction of edges in the CAG is irrelevant for the correctness of the formulation. However, the direction influences the form and number of edge constraints, and therefore has an impact on the performance of the generated 0-1 prob-

lem instance. An edge direction normalization step ensures that for any pair of arrays (a, b), all edges between nodes that represent a dimension of a and a dimension of b have the same direction, e.g., are all oriented "from a to b".

Let $SRC(b, a_i)$ denote the set of all nodes b_j that represent a dimension of array b and there is an edge from b_j to a_i. For each k, $1 \leq k \leq d$, and each non-empty set $SRC(b, a_i)$, IN-constraint of the form $\sum_{b_j \in SRC(b,a_i)} \; b\$a_{ik}^{jk} \leq a_{ik}$ are introduced. Let $SINK(a_i, c)$ denote the set of all nodes c_j that represent a dimension of array c and there is an edge from a_i to c_j. For each k, $1 \leq k \leq d$, and each non-empty set $SINK(a_i, c)$, OUT-constraint of the form $\sum_{c_j \in SINK(a_i,c)} \; a\$c_{jk}^{ik} \leq a_{ik}$ are introduced.

The total number of edge constraints is $\mathcal{O}(2d|E|)$, where E is the set of edges in the CAG. Each edge occurs in exactly two constraints, a IN-constraints of its sink node and a OUT-constraints of its source node. To be more precise, the number of edge constraints is the number of nonempty SRC and $SINK$ sets, multiplied by d. In the worst case, each SRC and $SINK$ set contains only one edge, resulting in $2d|E|$ edge constraints.

Objective function: A solution of the 0–1 problem formulation of the inter-dimensional alignment problem *maximizes* the following objective function under the given constraints:

$$\sum_{(a_i,b_j) \in E} \; \sum_{k=1}^{d} \; a\$b_{jk}^{ik} \;\; weight\,(a_i, b_j)\;.$$

The switch $a\$b_{jk}^{ik}$ is *on*, if and only if the corresponding edge is inside a partition. A solution maximizes the edge weights inside a partition and thereby minimizes the edge weights across different partitions.

Correctness of 0–1 Alignment Problem Formulation

Assume that the CAG may contain nodes of arrays with different numbers of dimensions. Each edge represents an alignment preference, weighted by its importance. The inter-dimensional alignment problem is a node partitioning problem such that (1) nodes that represent the same array cannot be in the same partition, and (2) the sum of the weights of edges between distinct partitions is minimal. The correctness criteria for a solution of the inter-dimensional alignment problem over the CAG are:

- *(C1) :* The solution is a node partitioning of the CAG, i.e., each node belongs to exactly one partition.
- *(C2) :* Nodes representing distinct dimensions of the same array must not belong to the same partition.
- *(C3) :* The partitioning of the nodes has the property that the sum of the weights of all edges between nodes in different partitions is minimal.

The correctness criteria *C1* and *C2* are enforced by the node constraints. The edge constraints must make sure that the cost of an edge is considered in the objective function, if and only if both, the source and the sink of the edge are in the same partition. By maximizing the sum

of all edge weights inside partitions, the sum of all edge weights crossing distinct partitions is minimized. Therefore, the correctness criterion *C3* holds. What remains to be shown is that the edge constraints formulation satisfies the following lemma:

Lemma Let $e = (a_i, b_j)$ be an edge in the CAG. For any solution of the 0–1 problem formulation the following holds:

$$a\$b_{jk}^{ik} = 1 \quad \text{if and only if} \quad a_{ik} = 1 \text{ and } b_{jk} = 1,$$

where k is the index of an arbitrary partition, $1 \leq k \leq d$.

Proof :
"$\Longrightarrow$": Assume $a\$b_{jk}^{ik}$ is switched on. Assume without loss of generality that the edge is directed from a_i to b_j. $a\$b_{jk}^{ik}$ occurs in exactly two constraints, namely the IN constraint for node b_j and partition k, and the OUT constraint for node a_i and partition k. The right hand side of the inequality constraints is b_{jk} and a_{ik}, respectively. Therefore, both a_{ik} and b_{jk} have to be switched on.

"$\Longleftarrow$": Assume a_{ik} and b_{jk} are both switched on. Assume without loss of generality that the edge is directed from a_i to b_j. $a\$b_{jk}^{ik}$ occurs in exactly two constraints, namely the IN constraint for node b_j and partition k, and the OUT constraint for node a_i and partition k. The right hand side of the inequality constraints is b_{jk} and a_{ik}, respectively. Due to the node constraints, all $a_{i'k}$ and $b_{j'k}$, $i' \neq i$ and $j' \neq j$, have to be switched off. This implies that all variables of the form $a\$b^{i'k}jk$ have to be switched off since they occur in OUT constraints of switched off variables $a_{i'k}$. Therefore, in the IN constraint for CAG node b_j and partition k,

$$\sum_{a_{i'} \in SRC(a,b_j)} a\$b_{jk}^{i'k} \leq b_{jk} \ ,$$

only $a\$b_{jk}^{ik}$ may be switched on.

Since the solution of the 0–1 problem determines the maximal solution of the objective function and since all edge weights are greater than zero, $a\$b_{jk}^{ik}$ must be switched on. Assume $a\$b_{jk}^{ik}$ is not switched on in an optimal solution of the 0–1 problem. Since a_{ik} and b_{jk} are both switched on, $a\$b_{jk}^{ik}$ can be switched on, resulting in a solution that is greater than the optimal solution. This is a contradiction.
□

Automatic Selection of Dynamic Data Partitioning Schemes Distributed-Memory Multicomputers

Daniel J. Palermo and Prithviraj Banerjee
University of Illinois at Urbana-Champaign

For distributed memory multicomputers such as the Intel Paragon, the IBM SP-2, the NCUBE/2, and the Thinking Machines CM-5, the quality of the data partitioning for a given application is crucial to obtaining high performance. This task has traditionally been the user's responsibility, but in recent years much effort has been directed to automating the selection of data partitioning schemes. Several researchers have proposed systems that are able to produce data distributions that remain in effect for the entire execution of an application. For complex programs, however, such static data distributions may be insufficient to obtain acceptable performance. The selection of distributions that dynamically change over the course of a program's execution adds another dimension to the data partitioning problem. In this paper, we present a technique that can be used to automatically determine which partitionings are most beneficial over specific sections of a program while taking into account the added overhead of performing redistribution. This system is being built as part of the PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project at the University of Illinois. The complete system will provide a fully automated means to parallelize programs written in a serial programming model obtaining high performance on a wide range of distributed-memory multicomputers.

Pattern-Driven Automatic Parallelization

Christoph W. Kessler, University of Trier, Germany

Sequential scientific programs operating on vectors and dense matrices usually have a nice property: they are largely made up of a comparably small set of typical operations, called "patterns", like elementwise array operations, reductions, simple recurrences, grid relaxations, basic linear algebra operations, etc.

We exploit this observation for the construction of a knowledge-based automatic parallelizer. A pattern recognition tool locally identifies occurrences of the patterns in a given sequential source program. These occurrences are replaced by calls to highly optimized parallel implementations with same semantics.

These parallel implementations can strongly differ from code produced by standard semi-automatic parallelizers since now complete algorithms can locally be exchanged by new, parallel ones. The parallel pattern implementations must be parameterized in problem sizes and array distributions of their operands. An array distribution engine determines a globally (near-) optimal data layout. To supply that engine with fast and reliable cost estimations, we exploit the pattern-driven approach by precomputing parameterized tables of execution times of the parallel implementations which are directly measured on the target environment.

In a first phase of a research project called PARAMAT, we have implemented the pattern recognizer and a test driver that outputs parallelized code for a massively-parallel shared-memory machine. The realization of a distributed-memory backend for PARAMAT is the issue of the second phase to be started in autumn of 1995.

The PARAMAT Project: Current Status and Plans for the Future

Christoph W. Keßler
Fachbereich 4 – Informatik
Universität Trier
D-54286 Trier, Germany
e-mail: kessler@ti.uni-trier.de

Abstract

Scalable parallel numerical libraries and automatically parallelizing compilers seem to be contrary approaches to the same goal: the user–friendly generation of efficient parallel numerical programs for shared–memory and distributed–memory multiprocessors.

We propose a framework that integrates the library approach and parallelizing compiler technology. It is based on fast and powerful pattern recognition in sequential source programs and considerate local algorithm replacement.

By a simplified prototype implementation, we demonstrate the functionality of this approach for a massively parallel shared–memory target machine, the SB-PRAM.

We further propose constructive guidelines to adapt the method to distributed–memory multiprocessors by integrating an automatic array distribution engine, by synthetic performance prediction, and by data–distribution–independent design of library routine specification.

Key words: Massively parallel computer, automatic parallelization, scalable parallel library, data distribution, pattern matching, PARAMAT system, SB-PRAM, Fork95

1 Introduction

It is a well–known fact that, unfortunately, parallelizing compilers have severe problems with generating efficient code for distributed memory target machines (DMS). This is due to several reasons: First, there are the NP–complete problems of finding suitable static [LC90] and dynamic [Kre93] distributions of the source program's arrays among the processors. Second, faced with today's distributed-memory systems, it gets more and more difficult for the compiler to predict the run time of a nontrivial parallelized program, due to caching, network contention, buffering and other hardware properties. Precise and quickly accessible run time information is, however, essential for global optimization of array distributions (e.g., [BKK94]). Third, a program that has been generated by a (semi-)automatic parallelizer by adapting the sequential code to given array distributions and inserting message passing instructions where nonlocal data may be required, must, in general, be transformed to perform efficiently on a DMS. However, it is hard for the compiler to select among the numerous optimizing transformations that are known today, from simple loop transformations up to local algorithm exchange. Furthermore, the compiler should be able to exploit hardware–supported higher–order communication primitives like broadcast, combine or reduce [LC91]. Maintaining interfaces to machine–independent message–passing libraries like P4 [BL92], PVM [LT93] or MPI [CGH94] may also be a source of inefficiency.

Thus, an access to expert's knowledge would be fine. One approach to exploit this idea has been followed during the last years with the development of parallel libraries for distributed-memory multiprocessors (e.g., [CDPW92], [FSSS92], [Thi93]; [KA93]) that can be used by the developer of a new program. We call a library of parallel routines *scalable* if the efficiency of each routine does, over a wide range, not primarily depend on the local problem granularity[1]. This will, in general, mean that in practice such a scalable routine contains several (parallel) algorithms for different magnitudes of local problem size.

Scalable libraries offer several advantages: The operands are explicitly given in a clean interface; this may make address calculations simpler. As the implemented algorithm of the library routine is known, the implementation can use explicit formulae for iteration sets, communication sets etc. [KMR90]. Overlapping communication with computation is generally possible. For library routines, the expense for assembler programming (which is highly required for many systems to get a maximum of node performance, see [Fri91]) is more profitable than for a user program which is used only once. Moreover, practically significant approaches to automatic array distribution (e.g., [BKK94]) require extensive knowledge-based compilation environments, e.g. precomputed run time tables.

However, scalable libraries for DMS have also disadvantages: First, a programmer must get used to a parallel library pack, and there has no standard been established yet. For dusty decks, the library approach is not applicable. Second, and even more important: There is no generally accepted way of how to obtain data distribution independence. Either, a library routine assumes or enforces a fixed distribution for each array operand (which makes programming of the routine easy, but severely limits efficiency), or the distributions of array operands must be passed to the routine (which makes efficient usage of the routine easy, but renders programming and interface more complicated), either explicitly as a parameter, or implicitly as a global variable. Data distribution independence will, if done right, result in a dramatical blow-up of library code size, since the parallel algorithms are usually *not* data distribution independent, and the different cases must be distinguished.

Furthermore, contemporary libraries for DMS often do not distinguish between multidimensional arrays and array access shapes like vectors or matrices; e.g., a matrix is assumed to be a two-dimensional array. We think that, although this case appears to occur frequently in practice, this is too restrictive. We rather regard a vector or a matrix as a 1D or, respectively, 2D data window onto a (potentially) higher–dimensional array.

Last but not least, such libraries are typically programmed in (message–passing–) C or Fortran, and thus the efficiency of the generated routine is limited by the quality of the compiler used, and so is the predictability of run times.

We offer an alternative to this classical library approach: We integrate the library, including extensive run time tables and handling of array distributions, into the compiler. Additionally, we offer a pattern recognition tool that fast and reliably detects occurrences of computations in the sequential source for which a parallel implementation routine exists [Keß94b] and replaces them by a call to this routine. This pattern–driven approach (named "PARAMAT", see [KP93, Keß95]) is motivated by the observation that programs that are reasonable candidates to be ported to DMS (especially, programs operating on vectors and dense matrices) are at a large degree made up of a relatively small set of typical, frequently occurring basic numerical operations ("patterns"). This observation has been supported by an empirical study of numerical algorithms and source codes [Keß94a]. [Keß94a] also contains the complete list of patterns.

In this paper, we will make clear that the integration of library and parallelizer can be realized

[1] We define *granularity* as the quotient of the overall problem size divided by the number of processors.

by our pattern–driven approach. After a short summary of PARAMAT in section 2, we propose overall guidelines for the construction of library routines (section 3), present an implementation for a shared memory target machine (section 4) and our plans for the specification of a data-distribution-independent and compiler-integrated scalable parallel library (section 5).

2 Pattern–driven automatic parallelization

Patterns are semantics–based, mostly data–parallel operations, such as e.g. elementwise vector and matrix operations, basic linear algebra operations, simple recurrences, grid stencils, relaxation operations, simple reduction operations, convolutions, shifts etc. A pattern is regarded to be a primitive with respect to mathematical properties, occurring data structures, memory access structure, array alignment preferences and run time behaviour. Regarding numerical operations on vectors and dense matrices, we have collected a list of 150 typical patterns (see [Keß94a]).

A *pattern instance* is generated if a piece of code has been recognized as an occurrence of a pattern. Unparsed, it looks similar to a call to an externally defined procedure, with the formal pattern operands ("slots") being bound to the corresponding program objects. Pattern instances can either be generated by our pattern recognition tool [Keß94b] or directly by the programmer (just bypassing the pattern recognizer).

Pattern recognition is robust against many typical code transformations encountered in sequential programs, such as loop distribution and interchange, loop blocking, loop unrolling and jamming, temporary variables. Exploiting the natural semantical interdependencies among the patterns makes the pattern recognition algorithm quite fast. More details on the pattern recognizer are beyond the scope of this report; they are published elsewhere [KP93, Keß94b, Keß94a]. For illustration, the following NAS Kernel

```
for (k = 1; k <= n; k++)
    for (i = 1; i <= l; i++) {
        c[i][k] = 0.0;
        for (j = 1; j <= m; j += 4)
            c[i][k] = c[i][k]  +  a[i][j]   * b[j][k]
                               +  a[i][j+1] * b[j+1][k]
                               +  a[i][j+2] * b[j+2][k]
                               +  a[i][j+3] * b[j+3][k];
    }
```

is recognized by the pattern recognition tool as an instance of Matrix–Matrix–Multiplication

```
_m=4*((m-1)/4)+4;
MM (i,k,j,c[1:l][1:n],a[1:l][1:_m],b[1:_m][1:n],0.000000);
```

within 0.3 seconds on a low–end SUN SLC including lexing and parsing the source and unparsing and printing the result. Many more results are listed in [Keß94a].

Due to the pattern instances, we know what the program locally does; thus we can infer additional knowledge on mathematical properties, efficient parallel implementations, their favorable array distributions and their run time behaviour on the target machine considered. The parallelization system can then easily use this knowledge to guide a sophisticated parallelization process including high–level program transformations up to local algorithm replacement.

3 Pattern implementations: organization and format

3.1 Code driver

The general structure of a PARAMAT back-end for distributed memory target machines has been outlined in [Keß95].

To access the output of the pattern recognizer, a *driver* for code generation is required (see Figure 1). This driver unparses the abstract syntax tree that has been constructed by the pattern recognition tool, while paying attention to directives from the PARAMAT user that may prevent it from selecting some types of implementations (see [Keß95]). The result is a text file containing the user program with a call to a (parallel) library routine inserted for each pattern instance, formulated in a high–level programming language (e.g., a parallel C or FORTRAN dialect).

For a *shared memory* target machine, we then simply can omit the handling of array distributions and message passing in the parallel implementations, which makes the whole system much simpler. The code driver just prints the matched abstract syntax tree and emits the pattern instances as calls to library routines, the parameters in a specific format (we will work out this in section 4).

It is important to note that the library code already contains the mapping of (implicitly formulated) parallel tasks to processors; thus we need not to consider any scheduling problems afterwards.

We however want to state the following key difference between (future) *distributed-memory* back-ends of PARAMAT and current scalable, data-distribution-independent parallel libraries for distributed-memory machines:

As PARAMAT performs a global optimization of array distributions at compile time, the final configuration of array distributions will be fixed already at compile time, and the parallel code induced by these distributions will be generated *after* that.

It follows that this code generation corresponds to a partial evaluation of the implementation specification at compile time. Thus, it would be less advisable to have a large library of parallel routines that have been made data distribution independent by accessing formal parameters for array distributions, with the final program being only managed by the linker. Instead, we should provide a parameterized (data distribution independent) *specification* of such parallel routines that can be read and partially evaluated by the compiler, but the compiler's output is a set of routines that are data distribution *dependent*.

The parameters of these output routines will thus contain no information on array distributions any more — thus, the format of parameters can be the same as that for shared memory routines. That is why we focus on the shared memory back-end in the sequel.

3.2 Parameter format

This subsection describes the general interface for passing vector and matrix parameters to PARAMAT back-end routines R. We distinguish between two cases, mainly depending on the data structure used to represent multidimensional arrays:

- the *short format* is used if for all vector or matrix operands accessing some array A, passed to that routine, the following holds:

  Array A of dimensionality $d > 1$ is implemented as a 1D array of pointers to consecutively stored subarrays of dimensionality $d - 1$, as suggested e.g. in [PTVF92] (thus the extents of the array axes need not to be known for addressing), and the origin of each vector or

matrix operand accessing A is equal to the origin of A, and the array dimensions accessed by the vector or matrix have unit access strides.

- the *general format* is used otherwise.

3.2.1 General format

To pass a vector v to a routine in general format, we must specify a parameter sequence `A, inc1, negv, absv`, where `A` is a pointer to the first element of the (sub-)array A that is addressed by v, `inc1` denotes the increment for v's accesses to A (which may also be negative!), `negv` is a flag that indicates whether the elements of A accessed by v should be negated before being processed further, and `absv` does the same for taking the absolute value. The extent n of v (i.e., the problem size) is passed once for all operands of a routine.

For a matrix m in general format, the parameter sequence is `A, inc1, inc2, TRANSPm, negm, absm`, where `A`, `inc1`, `negm` and `absm` are defined similar to the vector case above, and `TRANSPm` indicates whether the matrix access m to A is to be transposed or not, before being processed further. We must specify `TRANSPm` to keep the order of extents n_1, n_2 of m consistent for all operands of the routine.

In this case, the caller must know the array axis extents of A to specify the increments properly.

Example: The matrix access `-A[3][2:m][2][1:r][1]` to an array `A` with declared or allocated extents `A[k][m][n][r][s]` is passed as

```
(..., A[1+s*(1+r*(2+n*(2+m*3)))], n*r*s, s, 0, 1, 0, ...)
```

If inversions of vector or matrix operands should be required by the corresponding slot of the pattern instance, these are computed separately before the call, using a temporary variable.

3.2.2 Short format

The short format requires no specification of increments. A transposition flag is necessary, but, if applicable, computation of negation or absolute value of a vector/matrix is done separately before the call, as already done for inversions. Thus, parameter passing for a vector v involves only one parameter (the pointer to the first array element to be accessed), and for a matrix m only two parameters (the pointer and the `TRANSPm` flag).

The short format tries to avoid long library routines and eliminates some overhead involved in extensive parameter passing of the general format. The short format is justified since this "normal" case appears to occur quite frequently in practice.

3.2.3 Other parameters

Often, an initializing scalar, an initializing vector or even an initializing matrix is required by an operation, e.g. at matrix vector multiplication (MV), $y := Ax + y_0$, there is an initializer y_0 that may often be zero, but can also be an arbitrary vector access to some array (e.g., y). Accordingly, for Matrix–Matrix–Multiplication, the result matrix may be initialized by a scalar or by another matrix.

A PARAMAT routine that involves such an initializing operand, must be told whether it is a scalar `s`, a vector `v` (if applicable), or a matrix `m` (if applicable). Since the parameters are typed, we pass both possibilities, e.g. for MV (short format):

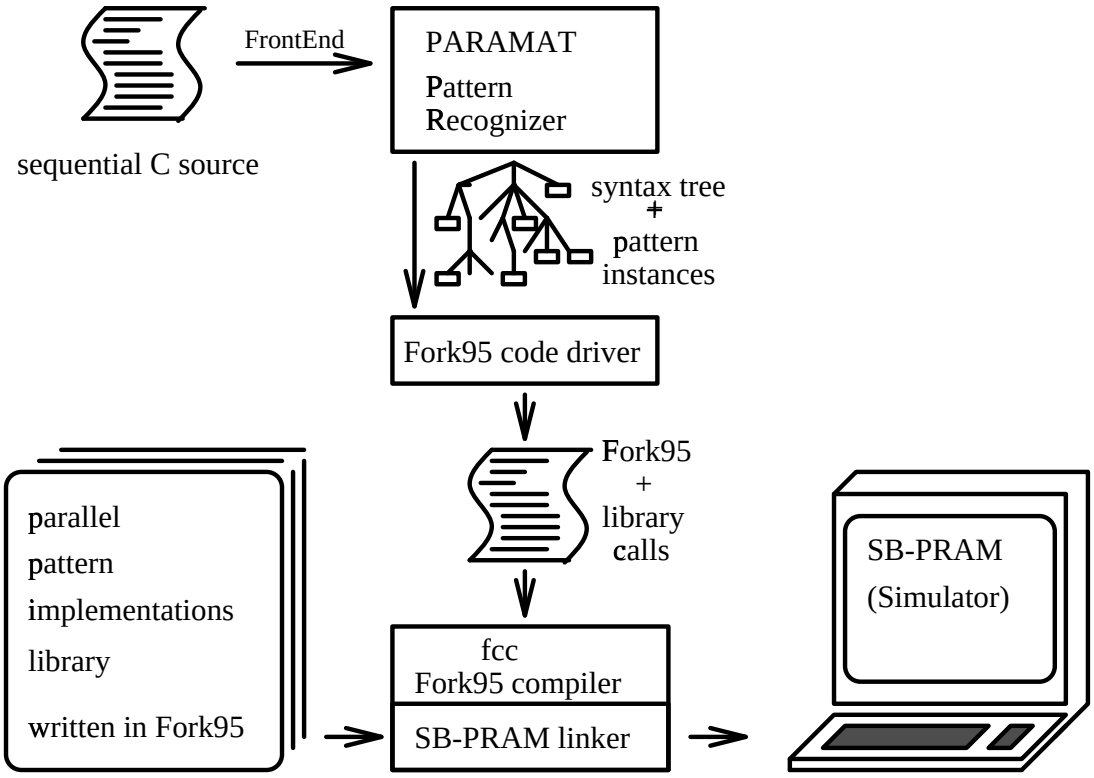


Figure 1: The overall structure of the shared-memory back-end for PARAMAT.

```
_MV( ..., s, v, ... )
```

The `_MV` routine initializes by `v` if `v` is not the null pointer; otherwise, it initializes by `s`.

4 A shared-memory back-end for PARAMAT

As a preparing step towards a future distributed–memory back–end for PARAMAT, we have built a shared-memory code driver and a library of scalable shared-memory implementations of the most important PARAMAT patterns.

We use this rudimentary back-end to demonstrate the functionality of the output produced by the pattern recognizer, and as a simple means for verification of the parallelized program, without having to build the whole system, especially, without having to care about array distributions.

As target machine, we use the SB-PRAM, a massively parallel shared memory multiprocessor with uniform memory access time. For simplicity of implementation, we have written the routines in a high-level language, Fork95 [HSS92, KS95] that we developed for the SB-PRAM in an other project.

The overall structure of the shared-memory back-end is given in Figure 1.

4.1 The target machine: SB-PRAM

The SB-PRAM [AKP90] is a lock-step-synchronous, massively parallel MIMD multiprocessor currently being built at Saarbrücken University, with up to 4096 RISC-style processing ele-

ments and with a physically shared memory of up to 2GByte with uniform memory access time. Because of its architectural properties, the SB-PRAM is particularly suitable for the implementation of irregular numerical computations, non-numerical algorithms, and database applications. Since the SB-PRAM hardware is not yet available (a 128-PE-prototype is to be expected for 1995), we use a simulator that allows to measure exact program execution times.

4.2 Library language: Fork95

Fork95 is a redesign of the PRAM language FORK. Fork95 is based on ANSI C and offers additional constructs to create parallel processes, hierarchically dividing processor groups into subgroups, managing shared and private address subspaces. Fork95 makes the assembler-level synchronity of the underlying hardware transparent to the programmer. It further enables direct access to the hardware-supplied multiprefix operations (integer only) that can be executed within 2 clock cycles.

The `start(`k`)` statement starts k processors simultaneously, with processor IDs called `$` numbered successively from 0 to $k-1$; if the expression (k) is omitted, then all `__STARTED_PROCS__` available processors are started. The processors started form one single processor group at the beginning, and execute synchronously (`sync` functions only) unless they enter the asynchronous mode via a `farm` construct. Within the `farm` body, synchronization is suspended; at the end of a `farm` environment, the current processor groups are synchronized explicitly. Throughout the whole program, the invariant is maintained that all processors in the same group operate synchronously, and all processors in the same group have a common shared address subspace.

The SB-PRAM offers hardware features that efficiently allow the user to partition the whole shared memory into private address subspaces (one for each processor) and a shared address subspace. Accordingly, variables are classified as either private (`pr`, this is the default) or shared (`sh`), where "shared" always relates to the processor group that defined that variable. An expression is private if at least one private subexpression may occur in it; worst-case assumptions (private) have to be made e.g. at dereferencing a pointer variable, as it is not statically known whether that variable will point to a shared or a private location at run time. Shared `if` or loop conditions do not affect the synchronity, as the branch taken is the same for all processors executing it. At an `if` statement, a (potentially) private condition causes the current processor group to be split into two subgroups: the processors for which the condition evaluates to true form the first child group and execute the `then` part while the other processors execute the `else` part at the same time. After that, the two groups are merged again by explicit synchronization of all processors of the parent group. The available shared stack of the parent group must be subdivided among the new child groups before the splitting, and joined afterwards. Similar subgroup construction is required also at loops with private exit condition. Thus, the processor groups form a tree–like hierarchy with the starting group at the root and with the current groups at the leaves at any time of the program.

To keep everything consistent, the compiler builds shared and private group frames at each group–forming statement. In order to avoid the overhead involved in this book–keeping, the user should do private computations as far as possible within asynchronous mode (inside a `farm` body or within `async` functions).

4.3 Example

4.3.1 Source and Pattern recognizer output

The following C code of a Conjugate Gradient solver has been derived from a FORTRAN77 source program that was supplied by Prof. A.K. Louis from the Numerics department of Saarbrücken University.

Beyond the (straightforward) translation from FORTRAN77 to C, we have already applied procedure inlining and constant propagation, two pre-transformations that are required by the pattern recognizer. Furthermore, we have omitted an outer while loop that was realized by a `GOTO` construct, the initializations and the I/O statements (because these are not yet included in the pattern recognizer).

```
main() {
   double a[100][100], x[100], b[100];
   int m, n, i, j, ire, im;
   double r[100], y[100], d[100], s, rm, alpha, beta;

   for (i = 0; i<m; i++) {
     y[i] = 0.0;
     for (j = 0; j<n; j++)
       y[i] = y[i] + a[i][j] * x[j];
   }
   for (i = 0; i<m; i++)
     y[i] = y[i] - b[i];
   for (i = 0; i<n; i++) {
     r[i] = 0.0;
     for (j = 0; j<m; j++)
       r[i] = r[i] + a[j][i] * y[j];
   }
   for (i = 0; i<n; i++)
     d[i] = - r[i];
   rm = 0.0;
   for (i = 0; i<n; i++)
     rm = rm + r[i] * r[i];
   for (im = 1; im <= ire; im++) {
     for (i = 0; i<m; i++) {
       y[i] = 0.0;
       for (j = 0; j<n; j++)
         y[i] = y[i] + a[i][j] * d[j];
     }
     s = 0.0;
     for (i = 0; i<m; i++)
       s = s + y[i] * y[i];
     alpha = rm / s;
     for (i = 0; i<n; i++)
     x[i] = x[i] + alpha*d[i];
     for (i = 0; i<m; i++) {
       y[i] = 0.0;
     for (j = 0; j<n; j++)
       y[i] = y[i] + a[i][j] * x[j];
     }
     for (i = 0; i<m; i++)
```

```
      y[i] = y[i] - b[i];
    for (i = 0; i<n; i++) {
      r[i] = 0.0;
      for (j = 0; j<m; j++)
        r[i] = r[i] + a[j][i] * y[j];
    }
    beta = 1.0 / rm;
    rm = 0.0;
    for (i = 0; i<n; i++)
      rm = rm + r[i] * r[i];
    beta = rm * beta;
    for (i = 0; i<n; i++)
      d[i] = -r[i] + beta * d[i];
  }
}
```

After the pattern recognizer has been run on this source program, the shared memory code driver produces the following output:

```
/* THIS FILE WAS AUTOMATICALLY GENERATED.  DO NOT EDIT IT! */

#include "fork.h"
#include "/home/pram/include/syscall.h"
#include "bpl.h"

sh float a[100][100];
sh float x[100];
sh float b[100];
 int m;
 int n;
 int i;
 int j;
 int ire;
 int im;
sh float r[100];
sh float y[100];
sh float d[100];
sh float s;
sh float rm;
sh float beta;

int main() {
  start
  {
 _MV (__STARTED_PROCS__,m,n,y,a,0,x,0.000000,0);
 _VAADD (__STARTED_PROCS__,m,y,b);
 _MV (__STARTED_PROCS__,n,m,r,a,TRANSP,y,0.000000,0);
 _VCOPY (__STARTED_PROCS__,n,d,r);
 _VQSUM (__STARTED_PROCS__,n,&rm,r,0.000000);
  for (im=1; im<=ire; im++)
  {
 _MV (__STARTED_PROCS__,m,n,y,a,0,d,0.000000,0);
 _VQSUM (__STARTED_PROCS__,m,&s,y,0.000000);
  {sh float _temp243 = 1.0/(s)*rm;
```

```
 _VAADDSV (__STARTED_PROCS__,n,x,_temp243,d);
  }
 _MV (__STARTED_PROCS__,m,n,y,a,0,x,0.000000,0);
 _VAADD (__STARTED_PROCS__,m,y,b);
 _MV (__STARTED_PROCS__,n,m,r,a,TRANSP,y,0.000000,0);
 beta=1.0/(rm);
 _VQSUM (__STARTED_PROCS__,n,&rm,r,0.000000);
 beta=rm*beta;
 _VADDSV (__STARTED_PROCS__,n,d,beta,d,r);
  }
  }
}
```

Manually adding initializers for `m`, `n` and `ire` and code for initialization of the vectors and matrices in this program (which could also be done automatically, of course) allows to execute this program for several problem sizes.

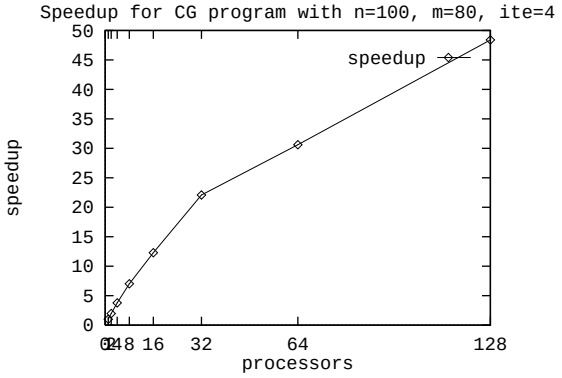
4.3.2 Library routines in Fork95

The library routines used here use the short format mentioned above. The routines are implemented straightforwardly and, of course, are still far from being optimal[2]; we show here two of them just for illustration (and as a byproduct, we would like to demonstrate how simple it is to write Fork95 programs):

```
sync void _MV (      /* short format of MV(2) matrix-vector product */
  sh int p,          /* #processors executing this call */
  sh int n,          /* length of lhs vector array access (problem size) */
  sh int m,          /* length of rhs vector array access (problem size) */
  sh vector x,       /* return value */
  sh matrix A,       /* n x m short format;  m x n if Atransp */
  sh int Atransp,    /* 1 if A transposed, 0 otherwise */
  sh vector b,       /* second operand vector, short form */
  sh float init,     /* the initialization value, if not a vector */
  sh vector initvec) /* the initialization vector, if not a scalar */
{
  pr int i, j;                                    /* private loop index */
  pr int slice = n/p + 1;                         /* local problem size */
  pr vector tmp;                                /*temp. accumulator vector */
  pr int lbi = $ * slice;                     /* lower bound of local slice */
  pr int ubi = lbi + slice;                   /* upper bound of local slice */
  farm {
    if (ubi > n)                                  /* adjust last slices */
       { ubi = n; slice = ubi-lbi; if (slice<0) slice=0; }
    tmp = (vector) malloc(slice);
    if (initvec)                                /* preset local slice: */
```

[2]For instance, we could exploit massive parallelism better if we would also admit parallel summation in `_MV` in a tree-like manner as soon as the number of available processors exceeds the extent of the parallelized array axis. This is also the reason why all library routines have been parameterized in the number of available processors. — Furthermore, we really recommend to write a performance–critical library in assembler, also on the SB-PRAM. There are some interesting hardware properties that cannot be fully exploited by the compiler, e.g. misusing push–instructions to move data, or getting rid of some worst–case–assumptions in the compiler. In this work, we have traded performance for programming time in order to rapidly get a prototype implementation running.

Figure 2: Speedup curve for the automatically parallelized CG program. Although the library routine implementations are far from optimal and written in a high-level language, and although the local problem granularity becomes quite small for larger numbers of processors, we obtain acceptable speedups even for a modest problem size of 100 x 80.




```
      for (i=lbi; i<ubi; i++) tmp[i-lbi] = initvec[i];
    else
      for (i=0; i<slice; i++) tmp[i] = init;
  }
  if (slice) {
   if (!Atransp)
    farm
      for (j=0; j<m; j++)  /* summing loop is (still) fully sequentialized */
        for (i=0; i<slice; i++)   /* innermost: parallel saxpy computation */
          tmp[i] = tmp[i] + A[i+lbi][j] * b[j];
   else                                                    /* transposed: */
    farm for (j=0; j<m; j++)
           for (i=0; i<slice; i++)
             tmp[i] = tmp[i] + A[j][i+lbi] * b[j];
  }
  farm for (i=lbi; i<ubi; i++) x[i] = tmp[i-lbi];    /* fill result vector */
}


sync void _VAADD (  /* short format of VAADD(1) accum. vector addition */
  sh int p,         /* #processors executing this call (synchronous!) */
  sh int n,         /* length of arrays _and_ array accesses (problem size) */
  sh vector a,      /* target and additive operand vector, short form */
  sh vector v)      /* operand vector, short form.     a[] += v[]    */
{
  pr int i;                                          /* private loop index */
  farm
    for (i=$; i<n; i+=p)          /*min(n,p) processors operate in parallel */
      a[i] += v[i];                  /* each PE computes up to n/p elements */
}
```

4.3.3 Results

For the CG program with 4 iterations and the problem sizes set to `n=100` and `m=80`, we obtain the speedup ratios given in Figure 2.

The speedup curve confirms that we could exploit massive parallelism with more than 32 processors better if we would, in this case, also parallelize the summing loop in `_MV`. The optimal ratio of processors along each matrix axis depends on many factors and can only be found out empirically. Such fine–tuning needs, however, to be done only *once*, namely at the generation of the library routines. Any program using these routines will then greatly profit from their

implementation. Maximal usage of these library routines also for sequential dusty deck sources can be enforced by the pattern recognition tool.

4.4 Conclusions

Although the simple back-end for the SB-PRAM is still in a preliminary state, we can already recognize some important points:

- The PARAMAT approach to automatic parallelization works fine for the SB-PRAM, even if the routines are written in a high-level programming language. Similar results for other shared memory target systems are to be expected.
- Scalability of all library routines is essential for exploiting massive parallelism also in shared memory systems.
- Fine-tuning of the library will result in increased speed-up, in particular for large numbers of processors.
- Writing library routines in a high-level language such as Fork95 is o.k. as long as pure performance is of only secondary importance.
- Following these guidelines, the classical library approach will be sufficient for shared–memory systems.

5 Towards a distributed–memory back-end

From these observations, we are able to derive guidelines for a future distributed–memory back-end of PARAMAT. The overall design of distributed-memory back-ends is given in Figure 3.

To simplify the system design being described in the sequel, we regard a given hardware environment as fixed; in particular, we regard hardware resources like number p, topology, and speed of processors, cache size and caching strategy, and memory size as constant. This corresponds to a 'dedicated' target machine. In the following considerations, we need therefore take no care any more for these hardware parameters. Nevertheless, scalability is still required since local problem granularity still depends on the problem sizes.

5.1 Synthetic performance prediction

A global optimizer determines the global distribution of all arrays occurring in the sequential source program. For optimization, this engine needs reliable run time figures to judge the quality of a specific distribution configuration. Classical bottom-up run time prediction of overall program run time given this distribution configuration is not advisable to do for larger application programs because of two reasons: First, because analytical, "low-level" performance prediction is unable to take care of typical hardware-specific effects like caching, buffering, network traffic etc.; second, because analytical performance prediction is generally too slow (e.g., the automatic array distribution engine of [DHR94] suffers mainly from slow analytical performance estimation). For this reason, we prefer "synthetic" performance prediction that assembles the global run time from table entries that have been generated off-line for each pattern, with one entry for each array distribution configuration and for each problem size magnitude (logarithmic scale). Such tables are required e.g. in [BKK94].

5.2 Automatic array distribution

A second requirement for on-line optimization of array distributions is that the parallel implementations are specified in a data-distribution-independent way. This may be done either by conditionals depending on the distribution parameters of one or several arrays, or by replication of parallel implementations, one for each possible distribution configuration. In each of these cases, it would be advisable to limit the possible distribution alternatives severely, instead of admitting arbitrary block-cyclic distributions of any block size. For vectors of length n, we allow the following distributions:

1. contiguous distribution (block size is n/p,
2. cyclic distribution (block size is 1) and
3. total replication (no distribution).

For a $m \times n$ matrix, we admit the following distributions:

1. contiguous row distribution (block size is mn/p, block shape is $(m/p) \times n$),
2. contiguous column distribution (block size is mn/p, block shape is $m \times (n/p)$),
3. cyclic row distribution (block shape is $m \times 1$),
4. cyclic column distribution (block shape is $1 \times n$),
5. contiguous quadratic blocks (block size is mn/p, block shape is $(m/\sqrt{p}) \times (n/\sqrt{p})$) and
6. total replication (no distribution, block shape is $m \times n$).

This limitation of array distribution alternatives is supported by the fact that for all our patterns [Keß94a], a locally optimal distribution for each array operand is contained in this list. We are aware of the fact that a *globally* optimal data distribution configuration may be made up of only locally suboptimal array distributions, although we believe that this scenario hardly appears in practice.

Quadratic contiguous block distributions are optimal for grid relaxation sweeps, since they minimize the surface-to-volume–ratio of the array partitions and thus the amount of data to be exchanged. In our framework, they are the only distribution scheme that distributes processors along more than one array axis. For quadratic distributions, however, we must add in this case the following constraint: The array (grid) A accessed by a matrix m must be 2–dimensional. Otherwise, imagine the following situation: Let A be three–dimensional, with axes A_1, A_2 and A_3, being distributed into quadratic blocks along, say, axes A_2 and A_3. Let m be a matrix access along the first and second axis of A. The number of processors along axis A_2 is $\sqrt{p}$, the number of processors along axis A_1 is 1 (not distributed). Thus, m has only $\sqrt{p}$ partitions, which limits parallelism unnecessarily, and, worse yet, the overall number of working processors is not constant any more for each call to the corresponding relaxation routine. Since we do not want to do everything twice, with one extra routine version for p and one for only $\sqrt{p}$ processors, we generally admit quadratic block distributions only for arrays of dimensionality equal to 2.

Problem sizes (corresponding to vector lenghts or matrix extents) need to be considered only in a specific interval $[N_{min}...N_{max}]$ of interest, e.g., from 8 to 16384. The parameter extent of that problem size axis thus contains $D = \log N_{max} - \log N_{min} + 1$ entries.

With these guidelines and with the limitation of array distribution alternatives given above, the parametrization space (and thus, the run time table size) for a pattern implementation with x vector operands, y matrix operands and z problem sizes contains $3^x \cdot 6^y \cdot D^z$ entries. For the MV matrix vector product above, we obtain an (uncompressed) table size of $54D^2$. Of course, this does not mean that we have to implement matrix vector product once for each of these configurations. Generally, several entries can be handled as a whole block, e.g., by taking array alignment relations [LC90, KLS90, KN90] into account, or ranges of problem sizes with similar run time behaviour. The run time tables can thus also be compressed according to this hierarchical parametrization structure of the parallel implementation.

5.3 Parameter format

The parameter format for distributed memory backend routines has not yet been finally determined so far. However, it seems clear that we will extend the general format for shared memory backend routines (see above) by adding array distribution parameters, e.g., one record, containing block shape and replication information, for each operand array of a routine, as e.g. done in the ScaLAPACK routines [CDWW94]. On the other hand, we can omit the first parameter indicating the available number of processors (see above).

5.4 Specification of parallel implementations

It remains the technical problem of how to code a parallel implementation in a data-distribution-independent way while maintaining explicit formulae for iteration and communication sets, and avoiding the overhead involved in evaluating complicated parametrization formulae at the target program's run time. We do this in two steps. First, PARAMAT specifies the parallel implementations in a target-machine-specific language (we favor C plus inline-assembler). This specification however allows complicated parametrization formulae or, if unavoidable, excessive replication of implementation code. Once the data distribution engine has determined a global distribution configuration for all array operands, we can derive the proper parallel implementation subroutines (comparable to those in the previous section) from that data-distribution-independent specification by partial evaluation (for reference, see [JGS93]) and dead code elimination. We obtain small and efficient message-passing–C sources that are data-distribution-dependent, and we need to extract only those routines from the specification library that are called by the matched user program. These are then compiled and linked together with the matched user program that has been produced by a suitable code driver (cf. Figure 3).

These routines extracted from the specification are also used to produce the run time tables. As this is a really tedious procedure, we plan to automatize table construction.

The realization of such a distributed-memory back-end is planned to be done in a research project starting in 1995 at Trier University, Germany.

6 Conclusion

We have presented a framework that integrates the scalable library approach and parallelizing compiler technology. It is based on fast and powerful pattern recognition in sequential source programs and considerate local algorithm replacement.

By a simplified prototype implementation using the Fork95 programming environment for the SB-PRAM, we have demonstrated the functionality of this approach for a massively parallel shared–memory target machine.

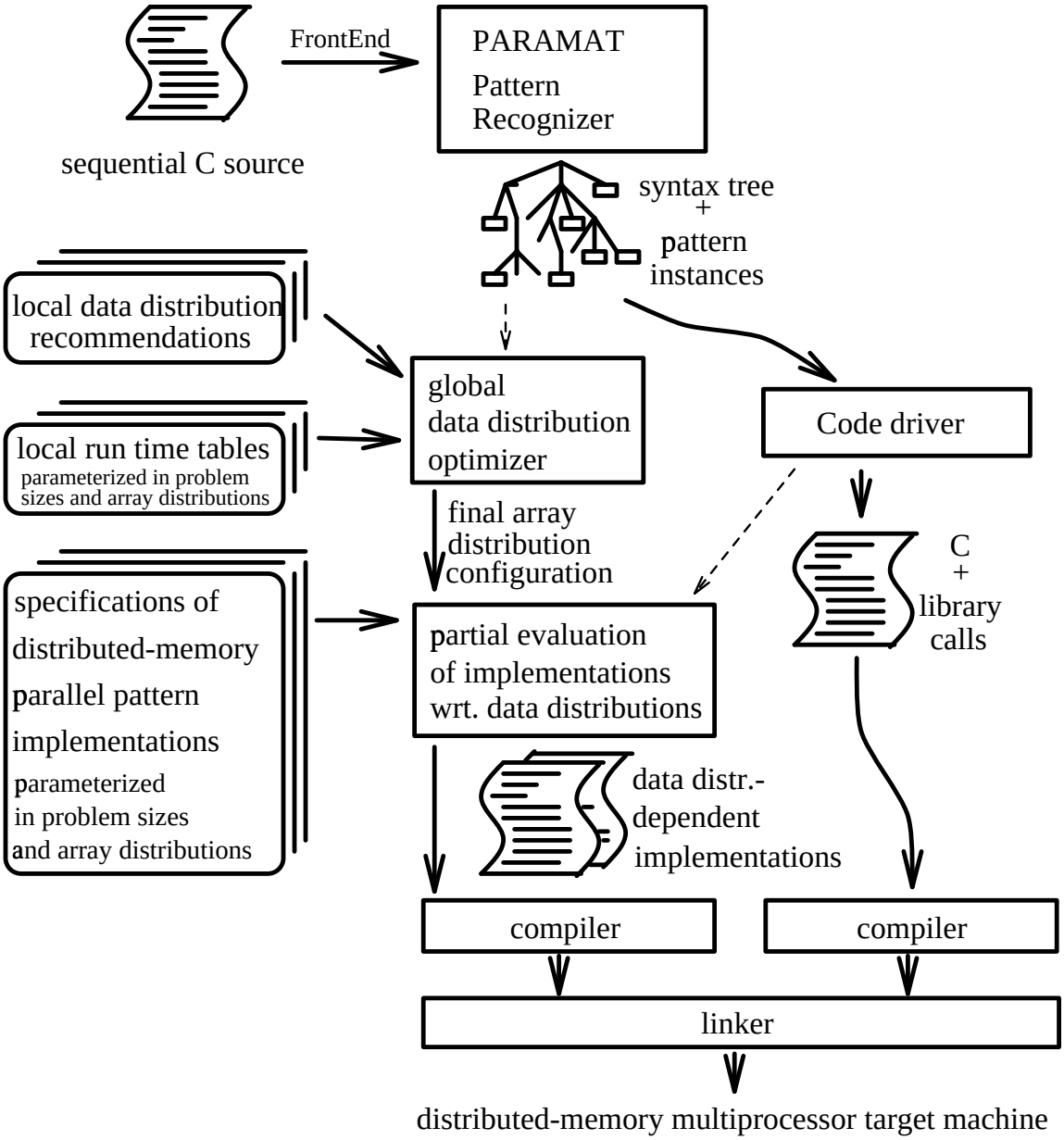


Figure 3: The overall structure of a distributed-memory back-end for PARAMAT.

We further have proposed constructive guidelines to adapt the method to distributed–memory multiprocessors, particularly by integrating an automatic array distribution engine and fast synthetic run time prediction. Extraction of data-distribution-dependent library routines is accomplished by applying partial evaluation to a data-distribution-independent specification of parallel implementations.

Acknowledgements

We thank Prof. A.K. Louis from the department of numerical mathematics at Saarbrücken University for providing the FORTRAN77 source code of the CG example program. The author further thanks the SB-PRAM system software team at Saarbrücken University for providing the hardware simulation environment and system software tools for our implementation.

Papers on PARAMAT, SB-PRAM, and Fork95

Information and papers on PARAMAT can be accessed via anonymous ftp from `alkoholix.cs.uni-sb.de` in directory `pub/PARAMAT`, and via www by `http://www-wjp.cs.uni-sb.de/paramat/index.html`. Information and papers on the SB-PRAM are accessible via ftp in `pub/sbpram`, and via www by `http://www-wjp.cs.uni-sb.de/ sbpram/index.html`. Regarding Fork95, look at ftp directory `pub/Fork95` and www page `http://www-wjp.cs.uni-sb.de/fork95/fork95.html`. The Fork95 compiler is available on `ftp.informatik.uni-trier.de` in directory `pub/users/Kessler`.

References

[AKP90] F. Abolhassan, J. Keller, and W.J. Paul. On Physical Realizations of the Theoretical PRAM Model. Technical Report 21/1990, Sonderforschungsbereich 124 VLSI Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1990.

[BKK94] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic Data Layout Using 0-1 Integer Programming. In *Int. Conference on Parallel Architectures and Compilation Techniques (PACT94), Montreal, Canada*, Aug. 1994.

[BL92] Ralph Butler and Ewing Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.

[CDPW92] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[CDWW94] J. Choi, J.J. Dongarra, D.W. Walker, and R.C. Whaley. SCALAPACK reference manual: parallel factorization routines (LU, QR, and Cholesky) and parallel reduction routines (HRD, BRD, and TRD) (Version 1.0 beta (December 31, 1993)). Technical Report ORNL/TM-12470, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, April 1994.

[CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland. Birkhäuser*, pages 21–1–6, April 1994.

[DHR94] Anne Dierstein, Roman Hayer, and Thomas Rauber. Automatic parallelization for distributed memory multiprocessors. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 192–217. Verlag Vieweg, 1994.

[Fri91] Stephen S. Fried. Personal Supercomputing with the Intel i860. *BYTE*, pages 347–357, Jan. 1991.

[FSSS92] R. Falgout, A. Skjellum, S. Smith, and C. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In *Proc. Scalable High Performance Computing Conference*, Apr. 1992.

[HSS92] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A High–Level Language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[KA93] Allan D. Knies and George B. Adams. The Integrated Library Approach to Parallel Computing. In *Proc. Scalable Parallel Libraries Conference.* IEEE Computer Society Press, Oct. 1993.

[Keß94a] Christoph W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, 1994.

[Keß94b] Christoph W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In *Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser Verlag AG, Basel, Switzerland, April 1994.

[Keß95] Christoph W. Keßler. Pattern-Driven Automatic Program Transformation and Parallelization. In *3rd EUROMICRO Workshop on Parallel and Distributed Processing, San Remo.* IEEE Computer Society Press, Jan. 1995.

[KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[KMR90] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 177–186, 1990.

[KN90] Kathleen Knobe and Venkataraman Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD machines. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 416–423, 1990.

[KP93] Christoph W. Keßler and Wolfgang J. Paul. Automatic Parallelization by Pattern Matching. In *Proc. of Second Int. Conference of the Austrian Center for Parallel Computation, Springer LNCS 734*, pages 166–181, Oct. 1993.

[Kre93] Ulrich Kremer. NP-Completeness of Dynamic Remapping. Technical Report CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Aug. 1993. see also: Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, Dec. 1993.

[KS95] Christoph W. Keßler and Helmut Seidl. Making FORK Practical. Technical Report 01/95, SFB 124 VLSI-Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken, 1995.

[LC90] Jingke Li and Marina Chen. Index Domain Alignment: Minimizing Cost of Cross–referencing between Distributed Arrays. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, 1990.

[LC91] Jingke Li and Marina Chen. Compiling Communication–Efficient Programs for Massively Parallel Machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

[LT93] Oak Ridge National Laboratory and University of Tennessee. Parallel Virtual Machine Reference Manual, Version 3.2. Technical report, August 1993.

[PTVF92] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition.* Cambridge University Press, 1992.

[Thi93] Thinking Machines Corp. The CMSSL Library (V. 3.1: Optimizing Numeric Operations Across High Volume Microprocessors, 1993.

Program Comprehension Techniques to Improve Automatic Parallelization

Beniamino Di Martino, University of Naples, Italy
Barbara Chapman, Universitiy of Vienna, Austria

Automated program recognition can play a crucial role in overcoming limitations of existing tools for the automatic parallelization of programs for distributed-memory architectures. Here the integration of parallelization techniques based on Program Comprehension into the Vienna Fortran Compilation System (VFCS) is described. After a pattern occurring in a program has been recognized it can be mapped automatically to a semantically equivalent code fragment for the target architecture. Such a mapping, which includes the selection of a data distribution, can often be done in a locally optimal way. The integration of this facility into the VFCS contributes to the objective of extending the limits of automatic parallelization, while maintaining the priority for generating high-quality target programs.

Program Comprehension Techniques to improve Automatic Parallelization

B. Di Martino[1] B. Chapman[2]

[1]Dept. of Computer Science and Systems
University "Federico II" - Naples - Italy
dimartin@cps.na.cnr.it

[2]Institute for Software Technology and Parallel Systems
University of Vienna - Austria
barbara@par.univie.ac.at

Abstract

Automated program recognition can play a crucial role in overcoming limitations of existing tools for the automatic parallelization of programs for distributed-memory architectures. In this paper, we describe the integration of parallelization techniques based on program comprehension into the *Vienna Fortran Compilation System* (VFCS). After a pattern occurring in a program has been recognized it can be mapped automatically to a semantically equivalent code fragment for the target architecture. Such a mapping, which includes the selection of a data distribution, can often be done in a locally optimal way. The integration of this facility into the VFCS contributes to our objective of extending the limits of automatic parallelization, while maintaining the priority for generating high-quality target programs. The paper discusses the implementation of a tool for recognition of parallelizable algorithmic patterns (*PAP Recognizer*) and outlines its integration into VFCS.

1 Introduction

Currently available tools for automatic parallelization perform a purely *structural* analysis of the sequential code. A semantic analysis (at a proper abstraction level), which could allow for *concept assigning* to pieces of code, and so the recognition of the algorithms that the code implements, is not performed. This kind of analysis could permit significantly more powerful transformations than those possible with existing techniques. Uses of a program recognition-based parallelization procedure could range from the automatic selection of a data distribution, via the automatic selection of sequences of optimizing transformations to the sequential code, via the code replacement with optimized parallel libraries, up to the automatic selection of the proper Parallel Paradigm for the given program (related to the target architecture), thus enabling much more flexible approaches to program parallelization than those provided by the SPMD paradigm. In this context, automated Program Recognition can play a crucial role in overcoming limitations of existing tools for the automatic parallelization of programs for distributed-memory architectures.

In this paper, we describe the implementation of a prototype tool that performs recognition of Parallelizable Algorithmic Patterns (*PAP Recognizer*), and how it has been interfaced with the *Vienna Fortran Compilation System*, a compiler which transforms a superset of HPF.

2 Program Comprehension and Automatic Parallelization

Program comprehension is the process of discovering abstract concepts in the input code and assigning them to their realizations within the code itself. This problem has been studied, for instance, in the context of reverse engineering to (partially) automate maintenance and reuse of sequential software. When the target is the recognition of concepts regarding the application domain, the process does not seem completely automatable [4]. However, some authors [5, 6] have argued that it can be automatable if only programming-oriented concepts, such as searches, sorts, structure transformations, numerical integration, etc., have to be recognized, and a number of approaches have been attempted in this context, with positive results.

Most of these approaches follow the methods of AI *Production Systems*, which can be roughly described as consisting of a *global database*, a set of *production rules* and a *control strategy*. Production rules operate on the global database, possibly modifying it; the control strategy selects applicable rules, recording already applied rule sequences; the computation ends when the global database satisfies a *goal condition*. These systems are generally *knowledge based*, i.e., they include *a priori* knowledge about the relevant domain in the global database.

Knowledge based approaches to Program Comprehension utilize *plan based* techniques. *Plans* are concept recognition rules which work in a hierarchical fashion, recognizing a concept when subconcepts satisfying a set of constraints have already been recognized [7]. Process recognition can therefore be viewed as a hierarchical parsing process.

Features identifying an (algorithmic) concept can be informally defined as the way some abstract functions are related and organized into a specific abstract control structure. By "abstract control structure" we mean structural relationships, such as control flow, data flow and calling relationships. It is worth noting that these relationships involve "abstract" objects, i.e. aggregates of variables or statements linked by a functionality. This set of structural relationships among abstract concepts defines an *algorithmic pattern*, also referred to as a *cliché* in [6]. Several approaches have been proposed according to this strategy.

Plan Calculus [6] represents code and clichés with graph structures, whose nodes correspond to subconcept instances, and whose arcs capture control and data flow relationships among them. Clichés recognition becomes a *graph parsing* process using a set of *graph grammar rules* to produce a *parsing tree* representing a hierarchical description of a plausible project of the program.

The PAT approach [5] uses an abstract, object-oriented representation for syntactic and semantic concepts composing the program, called *program events*. Each event is an instance of an event *class* and the classes are hierarchically structured. The plan's representation consists of a *path part*, describing the concept's components and a *test part* describing the constraints to be satisfied by components. An inferential pattern-directed engine derives new higher-level events from the existing ones, utilizing plans as inference rules.

In [10] we have argued that the ability to perform program recognition directed to algorithmic concepts turns out to be especially interesting with respect to parallelization, since the abstract concepts that should be recognized in order to apply effective parallelization strategies can be classified as programming-oriented.

In the same paper, the term *Parallelizable Algorithmic Pattern* (PAP) has been introduced to refer to a set of identifiable algorithmic properties characterizing a group of concrete algorithms that are parallelizable according to some common strategy.

Examples of Parallelizable Algorithmic Patterns can be found in the field of numerical computation: they range from low level concepts like vector shifting, scalar product, matrix-matrix multiplication, to high level concepts (algorithms) like methods for resolution of systems of linear equations (direct and iterative methods). These patterns are characterized by the fact that they

can be parallelized according to a common execution model, the Data Parallel one, or, more generally, the SPMD Paradigm.

With regard to this kind of patterns, the contribution of program comprehension to parallelization can be that of replacing the code implementing the recognized PAP by an optimized parallel routine performing the same task. Examples of these latter are the intrinsic and library procedures specified for HPF, or (at an higher level) the ScaLapack routines. The replacement of low level patterns with optimized sequential routines, like the BLAS, in each process, can be useful. Also, in the context of the SPMD paradigm, a suitable data distribution (or a set of possible data distributions, parametrized by architecture or run-time dependent parameters, as the number of processors or the problem size) can be automatically selected for PAPs; additionally, for some PAPs a sequence of standard unimodular transformations (loop skewing, tiling, ect.) can be selected, that can make the subsequent parallelization process executed by these parallelizers more efficient.

A more ambitious task for a PAP recognition-based parallelizer is the selection of an appropriate parallelization strategy, which could differ from the SPMD paradigm generally selected for HPF-like languages.

Alternatives to the SPMD model have been pointed out in the recent literature on parallel programming techniques, though in contexts other than automatic parallelization; the "parallel structure" of most programs can be classified according to a limited number of patterns, that can be viewed as representative of the basic ways to organize a parallel computation [1, 2]: this notion of parallel structure of a program can be informally defined as *the way processes forming the parallel program are created, synchronize and communicate, abstracting from the details of their sequential part*, and has been referred to with the term *Parallel Programming Paradigm*, or more briefly *Programming Paradigm* (PP). The availability of different PPs can be used to extend the capabilities of automatic parallelization tools, if they are provided with recognition facilities that can associate the proper paradigm to the code to be parallelized. As stated in [10], the algorithmic comprehension of the sequential code can be a driving feature in the task of selection of the proper PP: the notion of PAP has been introduced to clusterize algorithmic patterns that share the property to be parallelizable according to a common PP.

An example of a PAP based upon a paradigm different from the SPMD one is represented by the class of *Divide and Conquer* algorithms: examples are n-body problems, Branch and Bound algorithms and sorting algorithms. All of them share a divide and conquer resolution strategy, corresponding to the traversal of a (abstract) tree; the nodes of this tree are the subproblems obtained by dividing the parent problem, and the tree traversal represents the recursive operation of dividing and conquering the problem. Of course, the kind of problem, and so the actual nodes of the tree, depends on the particular algorithm.

As an example, the pseudo-code in Fig. 1a is an implementation of a quick sort algorithm, where a list is ordered by splitting and recursively ordering the sublists (the divide and conquer strategy of this algorithm, corresponding to the traversal of an abstract tree, is implemented in an iterative way). The (iterative) Divide and Conquer pattern is described in an informal way in Fig. 1b. The features that identify this algorithmic pattern concern the way some abstract functions (split problem, push a subproblem onto stack, pop problem from stack, set current problem to other subproblem, etc.) are related and organized into a specific control structure.

The key part of the Divide and Conquer pattern, the tree traversal, could be performed in parallel according to one of two Parallel Paradigms: *tree computation* and *processor farm*.

The first PP consists of a set of processes connected by communication channels according to a binary tree structure. Each process receives a problem to be solved, tests for a condition and then either splits the problem into two subproblems that are sent to two child processes, or does some processing and returns a result to its parent. When a process sends two subproblems,

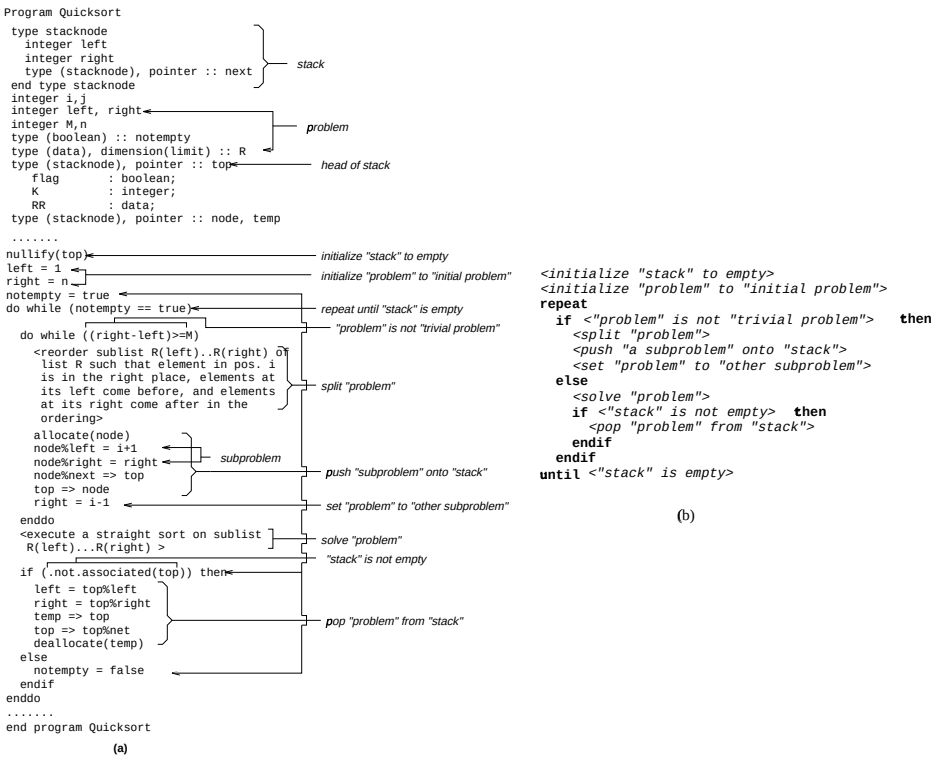
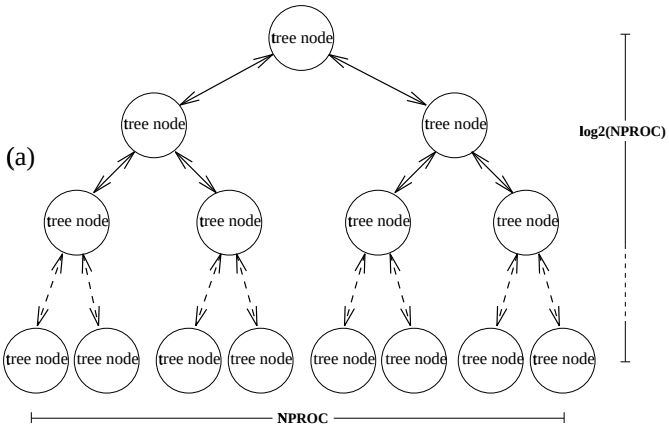


Figure 1: (a) Pseudo F90 code implementing a Quick Sort algorithm; (b) PAP of iterative divide and conquer.

it remains waiting for a reply from each child, then combines these replies and sends back the new result. In practice a number of questions (the depth of the tree of processes, if processes have to be created dynamically or the entire tree must be instantiated statically, etc.) have to be answered before a working program can be generated. However, they do not depend on the particular nature of the computation to be parallelized, but rather they are part of the PP and can be solved once and for all in the context of the paradigm itself. Figure 2 informally illustrates the paradigm through a graph specifying how processes communicate (Fig. 2a), and a pseudocode describing the behavior of the component processes (fig. 2b).

The second PP, *processor farm*, consists of a *coordinator* process and a set of *worker* processes that act as slaves of the coordinator. In a processor farm, the coordinator decomposes the work to be done into subproblems and assigns a different subproblem to each worker. Upon receipt of a subproblem, each worker solves it and returns a result to the coordinator. Again some details have to be defined before the PP can become a working program, and slightly different organizations can be selected for the processor farm (for instance workers may or may not be allowed to communicate with each other). However, even in this case, these issues do pertain to the PP definition and can be entirely dealt with in the paradigm context. The communication graph of the processor farm paradigm is illustrated in Fig. 2c. Which of these two paradigms to choose depends on the underlying architecture, but also on features of the particular divide and conquer algorithm, and so on the recognition of relevant features in the code implementing it. Indeed, if the problem decomposition phase is the costliest part of the algorithm, and so is better performed in parallel, the tree computation is more convenient. In any case, care has to be taken that subproblems are generated with nearly the same amount of associated work, leading to a well balanced tree computation that fits the process structure of the paradigm. Conversely, if the split phase is quite cheap, whereas the subproblems generated might correspond to very




```
proc tree_node:
  depth_max = log2(NPROC)
  if root node then
    depth = 1
    <prologue>
    <initialize "problem" to "initial problem">
  else
    recv depth from parent
    recv "problem" from parent
  endif
  if depth < depth_max then
    <split "problem">
    spawn a_son
    send depth+1 to a_son
    send "a subproblem" to a_son
    spawn other_son
    send depth+1 to other_son
    send "other subproblem" to other_son
    recv "a subresult" from a_son
    recv "other subresult" from other_son
    join "result"
  else
    <tree traversal>
  endif
  if root node then
    <epilogue>
  else
    send "result" to parent
  endif
end proc
```

(b)

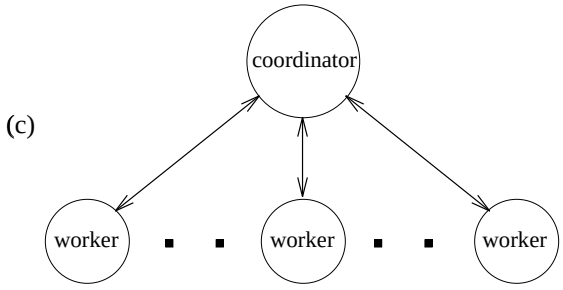


Figure 2: (a) Communication graph of the tree computation paradigm; (b) pseudo code of a tree node process; (c) Communication graph of the processor farm paradigm.

different amounts of work (owing to the presence of a pruning strategy to reduce the search space, for example), the processor farm paradigm appears to be best suited, since on the one hand splitting subproblems sequentially does not represent a significant penalty, and, on the other hand, it is quite easy to embed a load balance policy in the coordinator that can keep workers busy most of the time.

An important aspect of this approach to parallelization is that data distribution issues arising in the presence of irregularly structured algorithms, such as the one presented, can be handled in the framework of the paradigm selected. In the Quick Sort example, the data distribution is achieved by splitting the array of records to be ordered among processors: this problem is solved at run time, in a different way for each of the two suitable paradigms: in the Tree computation paradigm, each sublist is recursively splitted by each subprocess, and the resulting sublist are distributed to the spawned child processes; in the Processor Farm paradigm, the coordinator splits the overall data structure, and distributes it to the worker processes. A skeleton code that perform these tasks (an example being that in Fig. 2b) must be integrated with code implementing the concepts recognized, according to a general strategy that is described in the following section; this skeleton code can be described by suitable task parallel language features, such those provided by Fortran M [8] or Opus [9].

3 PAP Recognizer: a tool for Program Comprehension aimed to Code Parallelization

In [10] the overall architecture of a parallelizer based on PAP recognition is presented.

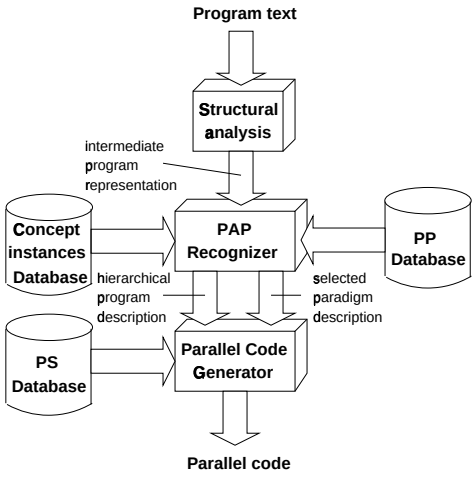


Figure 3: Architecture of a parallelizer based on PAP recognition.

The proposed architecture is reported in Fig. 3. The input program is first translated into a suitable intermediate representation, and control flow, data-flow and data-dependence analysis together with a number of normalizing transformations are performed on this representation. The intermediate representation is then analyzed by a *PAP Recognizer* and a hierarchical description of the program in terms of concept instances (stored in the *Concept Instances Database*) is produced, until a PAP (or alternative PAP's) is recognized. The recognition of a PAP selects the associated Parallel Paradigm (stored in the *PP Database*) as a possible parallelization strategy to be applied to the code (in some cases a set of PPs can be chosen, parametrized with respect to the underlying architecture, or to the problem size). Code parallelization is then performed starting from the description of the selected paradigm and the hierarchical description of the input program. The code generator builds the output parallel program relying on a *Parallel Skeleton* (PS) of the selected paradigm (stored in the *PS Database*).

A parallel skeleton is a generic implementation of a PP that needs to be instantiated to a concrete parallel algorithm in order to become an executable program. Instantiation of PSs consists of extracting, from the original input, those code segments corresponding to parts of the skeleton that have been left generic. In some cases this would require a certain degree of code transformation to integrate sequential code in a parallel, distributed memory framework. Once more, the importance of the recognition process is apparent: indeed, if some skeleton requires nontrivial transformations for sequential code integration, the corresponding information can be added to the paradigm description stored in the PP database; this information may help the recognizer to identify just those particular concepts that are needed to correctly perform the mentioned transformations. Another feature that could turn out to be particularly useful in the code generation phase is the hierarchical description produced by the recognizer: if the links between increasingly abstract concepts of the hierarchy and sequential code implementing them are properly represented, sequential code restructuring needed during code generation will be greatly simplified.

It is worth noting that all the observations made so far do not imply any assumptions about the actual nature of PS representation in the PS database. It may be just an abstract operational description of the paradigm that has to be translated into actual executable code during instantiations, or a compiled library that need only be linked with problem specific code derived from information produced in the recognition phase, or something in between. The source code may be described by suitable language features, such as those provided by Fortran M [8] or Opus [9].

These issues, however, are beyond the scope of this paper.

This paper focuses on the PAP recognition phase, and its integration within an SPMD oriented parallelizing compiler, the Vienna Fortran Compilation System [3], to provide more powerful automatization capabilities to the parallelizing compiler. A prototype of the PAP Recognizer is being developed for integration into the Vienna Fortran Compilation System. In this section we give details about the implementation of the PAP Recognizer prototype. In the next section we will discuss how the recognition phase can be integrated within the Vienna Fortran Compilation System.

First we describe the Internal Program Representation, as built and utilized by the PAP Recognizer; after that, we give details on the implementation of this representation, and of the recognition process, with reference to the system shell utilized, i.e. Prolog.

The output of the structural analysis phase, that is syntactic, control flow, data flow and data dependence informations on the program, is utilized to build the *Base Internal Representation* of the program, which is stored in the *Concept Instances Database* in the form of *base concepts*.

This base level of the representation is substantially a Program Dependence Graph (PDG), whose nodes represent statements and whose edges represent control and data dependences. This PDG structure slightly differs from a standard PDG graph, in that it is augmented with the following structures:

- control dependence edges are labeled with the branch (true, false) of the dependence, and data dependence edges are labeled with the dependence variable identifier and the kind of the dependence (loop independent or loop carried);
- assignment statement nodes are augmented with two tree structures representing its left and right hand sides; subscript expressions of array variable instances are likewise represented by tree structures, linked to the node representing the array variable instance;
- each control statement node is augmented with the tree structure representing the control conditional expression;
- each node of the augmented PDG points to the corresponding nodes of the syntax tree, so that a direct reference from the abstract concept recognized to the code implementing it is possible; in this respect, the PDG could be viewed as a kind of WEB;
- variable definitions are explicitly represented too, including information relative to the type, the dimensions for array variables, and including pointers to the corresponding nodes of the syntax tree.

The overall *Internal Program Representation* is generated during the recognition phase performed by the PAP Recognizer. It has the structure of a *Hierarchical PDG* (HPDG), reflecting the hierarchical strategy of the recognition process. The base layer of the HPDG describes the program at the statement and operator level. Starting from the base internal representation, the recognizer performs the concept parsing, following the rules specified by the plans (see details after in this section). As long as the parsing process proceeds and more and more abstract concepts are recognized, they are represented as nodes in increasingly higher layers of the HDPG. More specifically, each node representing a concept is linked to the nodes of the lower layer representing subconcepts. Control and data dependence edges for the newly created abstract concept nodes are inherited from those of the composing subconcept nodes, in a way that is characteristic for each concept, and which is determined by the plan which recognizes it. Internal Program Representation is stored in the *Concept Instances Database* which is incrementally updated by the PAP recognizer as the recognition advances.

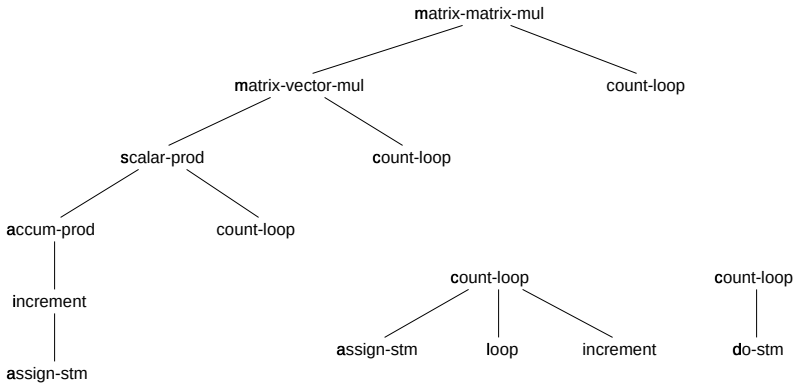


Figure 4: Hierarchical structure of matrix matrix multiply PAP.

The PAP Recognizer prototype relies on Prolog as a system shell. This choice has been mainly driven by its capability to be utilized as a deductive inference-rule engine to perform the hierarchical concept parsing.

The base internal representation is represented in the Prolog environment by a set of *base facts*, which describe all information related to the syntax (identifiers, base data-structures), to the control flow (control dependences) and the data flow (data dependences): facts which represent variable declarations and definitions, statements, and expressions in statements, correspond to the nodes of the base PDG and associated attributes; facts concerning control dependence and data dependence among statements, represent the arcs of the PDG.

The overall internal representation is represented in the Prolog environment by a set of facts, asserted during the recognition process, which represent the abstract concept nodes of the HPDG. To deal with the hierarchical structure of recognized concepts, compound terms describing them have been used; this permits the inspection of the subconcepts composing a concept, at every level; also, it allows the representation of the hierarchical description of recognized concepts in a very easy way, and with links to the syntax tree, that is to the user's code.

We give now a description of the recognition process using, as a case study, the matrix-matrix multiply algorithmic pattern. Even though very simple, this pattern may be implemented according to different PPs, requiring an appropriate data distribution strategy.

Figure 4 presents the hierarchical structure of this PAP, which is composed by several subconcepts, related, other than by the hierarchy of composition, by data and control dependency constraints (not shown in the figure). Figure 5 illustrates the plans for recognition of the matrix-matrix multiply PAP, and some of the concepts composing it. We follow, for the specification of the plans, the abstract formalism presented in [7]. The specification includes the components, or subconcepts of a concept, and constraints on and among subconcepts. As can be seen by these specifications, components are not related by constraints regarding the syntactic properties of the code (such as textual nesting of components), but rather "semantic" properties like control and data dependence relations.

Plans for the recognition of concept instances are implemented by Prolog clauses: the head of the clause represents the concept to be recognized, the body of the clause is composed by terms representing the composing subconcepts, and constraints among them.

The Prolog inference mechanism allows the matching of each of the terms in the body of a clause (the composing subconcepts or the constraints of a plan) with the head of one of the clause representing the plan to recognize that concept, or to check the satisfaction of that constraint.

The described approach to algorithmic pattern recognition permits successful handling of a

```
plan MATRIX-MATRIX-MULTIPLY ( ARRAY1: @array1ident,
                              SC-PR-POS-IN-ARRAY1: @scPrPosInArray1,
                              MM-POS-IN-ARRAY1: @mmPosInArray1,
                              ARRAY2: @array2ident,
                              SC-PR-POS-IN-ARRAY2: @scPrPosInArray2,
                              MM-POS-IN-ARRAY2: @mmPosInArray2,
                              RESULT-ARRAY: @resArrayIdent,
                              MM-POS-1-IN-RES-ARRAY: @mmPos1InResArray,
                              MM-POS-2-IN-RES-ARRAY: @mmPos2InResArray)
consist of
  matr-vec-mul: MATRIX-VECTOR-MULTIPLY(ARRAY1: @array1ident,
                              SC-PR-POS-IN-ARRAY1: @scPrPosInArray1,
                              MV-POS-IN-ARRAY1: @mmPosInArray1,
                              ARRAY2: @array2ident,
                              SC-PR-POS-IN-ARRAY2: @scPrPosInArray2,
                              RESULT-ARRAY: @resArrayIdent,
                              MV-POS-IN-RES-ARRAY: @mmPos1InResArray)
           loop: COUNT-LOOP(CNT-VAR: @indexIdent, INIT-VAL: @@, LAST-VAL: @@, STEP-VAL:1)
such that
          scalar-prod: SCALAR-PRODUCT(@@)
                       PARENT(scalar-prod,matr-vec-mul)
    sc-pr-update-accum: ACCUM-PROD(@@)
                       PARENT(sc-pr-update-accum,scalar-prod)
                       CONTROL-DEP(sc-pr-update-accum,loop,true)
              mv-loop: COUNT-LOOP(@@)
                       PARENT(mv-loop,matr-vec-mul)
                       CONTROL-DEP(mv-loop,loop,true)
                       MEMBER(@SubScrExpArr2,@SubScrListArr2,@mmPosInArray2)
                       ANCESTOR(@indexIdent,@SubScrExpArr2)
                       MEMBER(@SubScrExpResArr,@SubScrListResArr,@mmPos2InResArray)
                       ANCESTOR(@indexIdent,@SubScrExpResArr)
```

Figure 5: Plan for recognition of matrix-matrix multiply concept instances.

number of problems arising in the context of Program Comprehension. These have been described in [6], and can be summarized as:

- *syntactic variation*;
- *delocalization*: the implementation of a concept can be spread throughout the code;
- *implementation variation*: an abstract concept can be implemented in many different ways, and so has to be represented by several and alternative clichés;
- *overlapping implementations*: often the implementations of two or more distinct concepts are merged, so portions of a program may be part of more than one cliché.

The use of a rule-based system as Prolog, and the specification of (inter-statement level) patterns by control and data dependence relations, play a crucial role in overcoming the above mentioned difficulties.

The syntactic variation problem is solved by characterizing inter-statement level concepts with non-syntactic properties like control and data dependence: in this way the "semantic" features of concepts are exploited, instead of the characteristics of their implementations, like the use of particular constructs, or the textual nesting of components.

The delocalization problem is solved by the unification mechanism of Prolog, that attempts to match all the facts in the database with the active rule, ignoring the order of the statements, represented by these facts, in the program.

The implementation variation problem is solved by the backtracking feature of the unification mechanism, because it provides for the specification of a concept by multiple plans, for the application of a plan to all possible instances of its subplans (and all possible constraints among them), and for the recognition of several instances of the same concept.

The overlapping implementation problem is solved by the fact that the Prolog mechanism does not restrict the use of facts to one rule: all facts currently present in the database are available for firing new rules, even if these were used in a previous rule firing.

4 Integration of the PAP Recognizer within VFCS

In this section, we will discuss how the recognition phase can be integrated within the Vienna Fortran Compilation System.

Referring to the scheme in fig. 3, the structural analysis is performed by the VFCS front-end, whereas the parallel code generation is carried out by the VFCS back-end.

During the initial analysis phase, code is first normalized by the VFCS front-end. Control flow, data flow and data dependence analysis is then performed. The output of this phase, together with the syntax tree produced by the parser, are utilized to build the Base Internal Representation of the program, described in the previous section. To provide a link from Prolog facts to the code, the nodes of the syntax tree, generated by the VFCS front end, have been labeled. The base prolog facts describing syntax information are produced by a traversal of the syntax tree. These facts represent nodes and attributes of the base PDG, as described in the previous section. Consulting suitable VFCS representations (the control flow graph and the DU chains), facts concerning control dependence and data dependence among statements are produced, which represent the arcs of the PDG; attributes based on the control dependence facts are the branch in which there is a dependence, and attributes based on the data dependence facts are the identifier of the variable affecting the dependence and the kind of the dependence (loop independent, loop carried, the level).

Output from the PAP Recognizer to the back end of VFCS is the hierarchical description of the recognized PAP, in the form of a tree structure with links to the syntax tree as terminal nodes, together with the selection of the associated Parallel Paradigm, as possible parallelization strategy to be applied to the code.

As we have already mentioned, in the context of the VFCS parallelization procedure the PP selection can determine the selection of a data distribution strategy that matches both it and the features of the target parallel architecture, and, in some cases, can also drive the application of specific optimizing transformations that could substantially improve the subsequent parallelization steps. In this case, program restructuring is as follows. The PAP description provides the links to the syntax tree nodes representing declarations of the data structures (arrays) which are candidates for distribution, together with the kind and the dimensions of the distributions selected; the syntax tree can be augmented with the Vienna Fortran nodes representing the selected data distribution, and the resulting Vienna Fortran program can be processed by the VFCS. As a second possibility, PP selection can drive the replacement of the fragment code implementing the recognized PAP by an optimized parallel routine. In this case, the hierarchical description of the recognized PAP turns out to be particularly useful: code substitution can be performed in a controlled way at different abstraction levels, and information about the functions implemented by any particular code fragment can be easily accessed. As an example, it is possible to substitute the code implementing the PAP in its entirety, or also the code implementing a subconcept of it. As another example, variables which shall replace the formal parameters of the optimized routine, can be localized in the code looking at their functionality in the context of the PAP.

Of course, the selection of a suitable data distribution for the recognized PAP, besides being dependent on the amount of work it involves and on the target architecture, is related to the distribution choices for the surrounding code of the PAP. This may lead to distribution conflicts. This situation is managed by our approach as follows: VFCS generates a Component Alignment Graph, along the lines proposed in [14] and refined in [15], for code not recognized; by using the Weight Finder [16], an advanced profiler for Fortran programs also integrated into VFCS, we can estimate the weight of the recognized PAP in relation to the overall computation. If it is bigger than a machine-specific threshold, the best data distributions for the best realization

of the PAP are selected (depending on the target architecture), and they drive the selection of the pending distributions. In the other case, the strategy is the opposite one: the distributions suggested by the Component Alignment Graph and an analysis of subscript expressions, drive the selection. Among all the possible realizations of the PAP (whose descriptions are stored in the PP database), the realization that best fits with the suggested distributions is chosen.

5 An example

We show now how a number of instances of a simple parallelizable algorithmic pattern, the matrix-matrix multiplication, are recognized in code fragments.

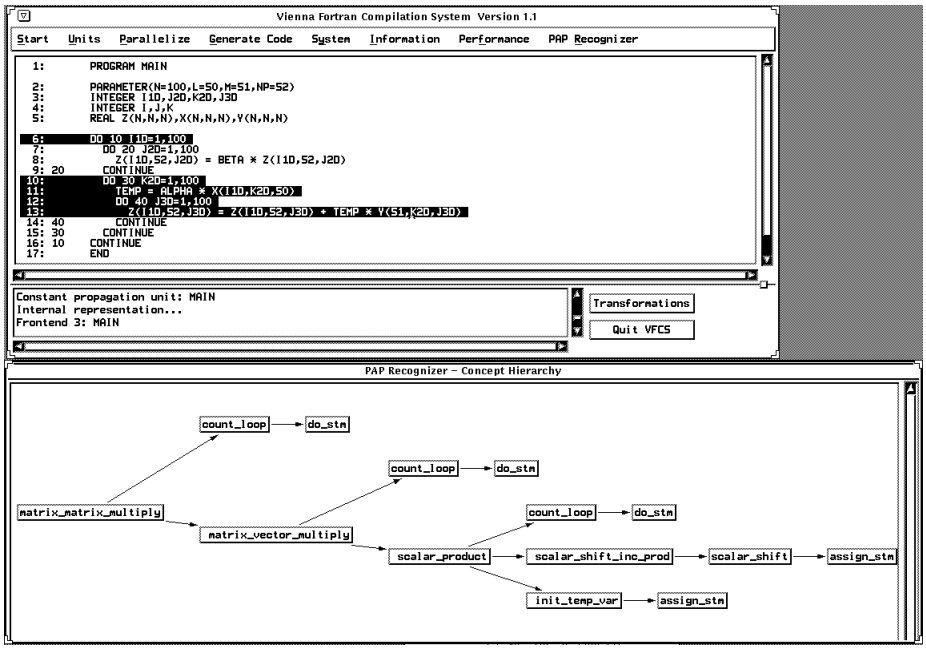


Figure 6: First example: visualization of the code implementing the overall matrix-matrix multiplication PAP.

Fig. 6 (upper window) shows a first Fortran program segment implementing matrix matrix multiplication, normalized by the VFCS front end. This version is analyzed by the PAP Recognizer, and Fig. 6 (lower window) shows the hierarchical structure of the recognized pattern, in the form of a graph. By clicking on each of the graph nodes, representing a composing subconcept, it is possible to highlight the corresponding code implementing it, as is shown in Fig. 6 for the overall concept instance, and in Fig. 7 and 8 for some composing concepts.

In Fig. 9 we present a second instance of the matrix matrix multiplication algorithm, with a temporary accumulator variable for the scalar product subconcept.

As can be seen in these figures, the tool is able to recognize the concept even in the presence of a temporary scalar variable, and of a loop interchange. The matrix multiplication factor arrays and resulting array are also recognized, as are the subscript positions involved in the inner product and in the overall multiplication steps.

With regard to the code generation, this pattern permits several possibilities. In the context of the SPMD paradigm, there are several possible data partitions of the arrays involved, whose optimal application depends on their dimensions, and on possible conflicts deriving from their use in the surrounding code. These conflicts are taken into account by evaluating information

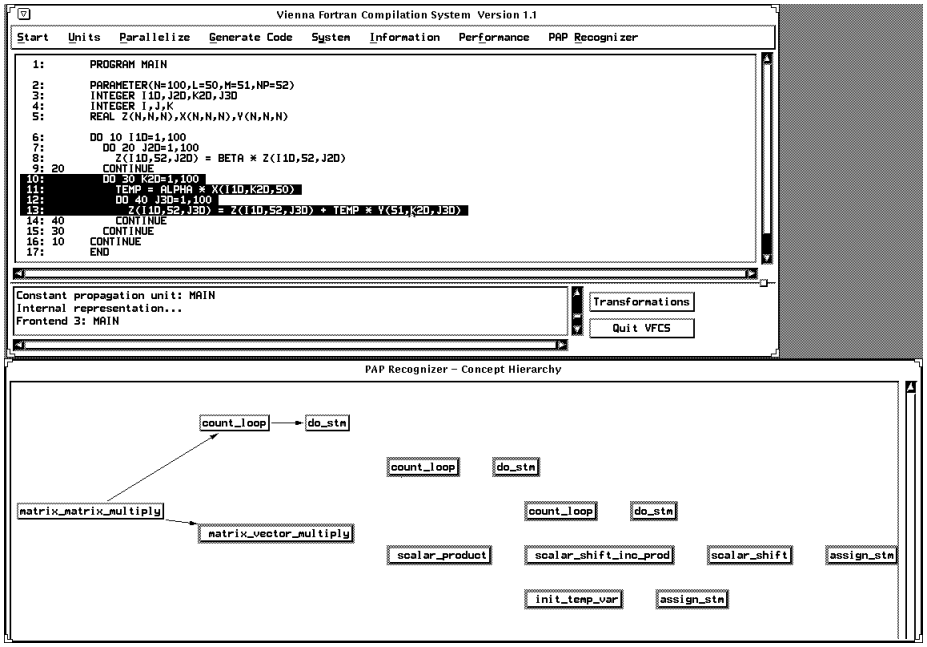


Figure 7: First example: visualization of the code implementing the matrix vector multiplication subconcept.

from the Weight Finder and the Component Alignment Graph, as described in the previous section. After choice of data distributions, and the modification of the syntax tree for the production of the VF program with distribution annotations, the parallelization process of the VFCS is applied. After that, the sequential code of the node processes can be restructured by replacing the code fragment implementing the PAP with an optimized sequential routine: the overall fragment code can be replaced by the BLAS3 GEMM routine, or the fragment code implementing the subconcept inner product, can be replaced by the BLAS1 DOT routine. This is an example of the usefulness of the hierarchical PAP description, to perform code substitution at different abstraction levels. In the context of paradigms different from the SPMD one, the pipeline algorithm proposed in [17] can be applied. An MPI implementation of it has been described in [18].

6 Implementation status

A first version of the PAP Recognizer is operational, and able to recognize a few dozen concepts related to numerical computation (these will be specified in a forthcoming Technical Report [11]).

With regard to the integration to the VFCS, the input interface, that is the automatic construction of the base internal program representation from the VFCS front end representation, has been implemented. With regard to the output interface, we have implemented a module for the graphical visualization of the PAPs hierarchical description, where the user may select subconcepts from the graph, and the code fragment they represent is highlighted.

The full system will target runtime features from both the Nexus [12] and Chant [13] runtime systems which provide functionalities to create and synchronize tasks as needed to generate code according to several possible paradigms.

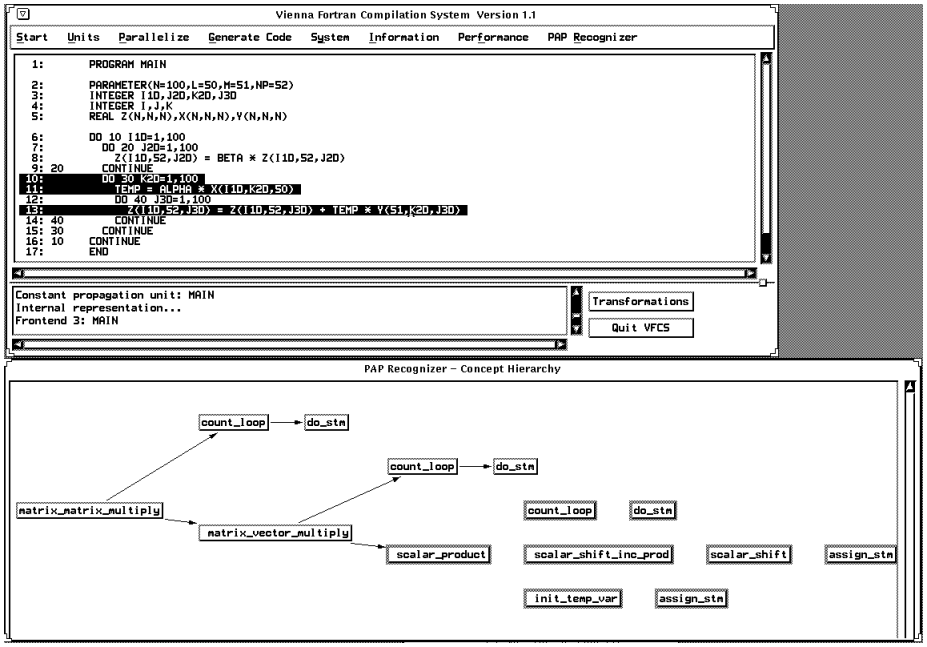


Figure 8: First example: visualization of the code implementing the scalar product subconcept.

7 Related work

We are acquainted with two other efforts aimed at application of program understanding techniques to automatic parallelization: work of Kessler and Paul [19], and proposal of Bhansali, Hagemeister, Raghavendra and Sivaraman [20].

The first effort proposes the application of pattern matching techniques for replacement of code implementing recognized patterns (mainly basic numerical computations) with optimized parallel libraries. Pattern matching process consists in a traversal of the syntax tree (augmented with arcs representing data dependences among statements) which performs the bottom-up matching of the code with the set of patterns, triggered in a hierarchical way. Differences with our approach are in the aims, and in the techniques applied for program understanding. With regard to the approach to the recognition, Kessler's approach perform a "syntactic" pattern matching, in the sense that it is performed on the base of a traversal of the syntax tree. Features of the code at a "semantic level", like control and data dependence, are not taken in account explicitly as concepts features (data dependence is used only implicitly, to drive the traversal of the syntax tree). Moreover, a leading feature for the matching is the nesting depth of array accesses within do-loops. We argue that this approach is weak with regard to non-locality, and syntactic variations of concepts implementation in the code, compared to our approach that is not based on syntactic features like the control environment or depth of nesting, but on more semantic ones like control and data dependences. Moreover Kessler's approach doesn't allow backtracking on pattern matching; it is a strong point for speed, but a weakness for the recognition. With regard to the aims, Kessler's approach will perform, in the area of numerical computation, replacement of code with optimized libraries; our approach will perform, in this area, selection of data distribution and optimizing transformations, before of the application of well known techniques of automatic parallelization; moreover, our approach is targeted to the selection of parallel paradigms other than the SPMD one, and so not only in the area of numerical and data parallel applications.

The second effort [20] is aimed at the use of abstract pattern matching techniques to provide

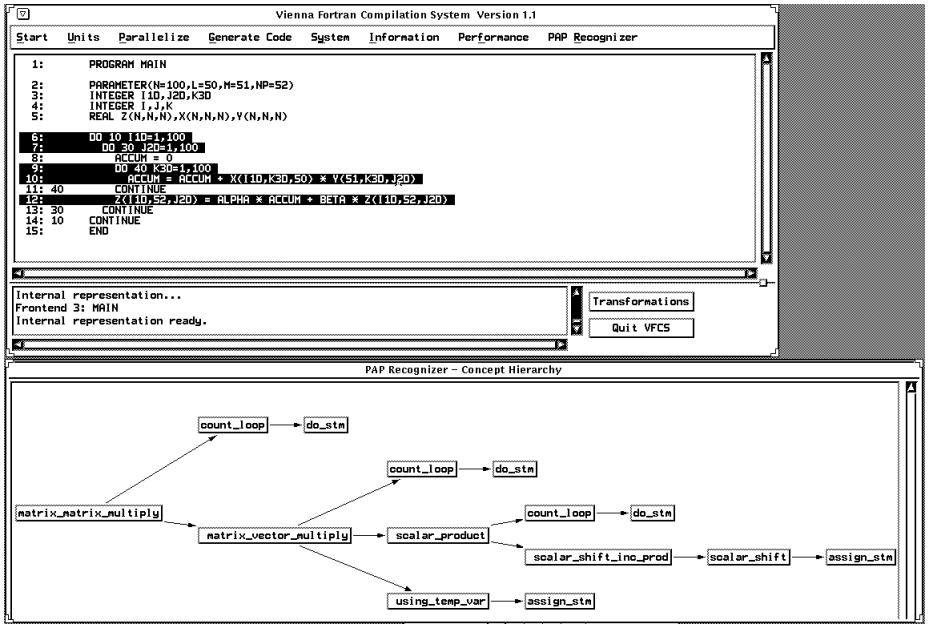


Figure 9: Second example: visualization of the code implementing the overall matrix-matrix multiplication PAP.

to the user suggestions of possible replacements of recognized patterns by efficient parallel code; it is still, of what we are aware, at an early stage, without a working prototype or a well defined comprehension technique.

References

[1] P. Brinch Hansen, "Model programs for computational science: a programming methodology for multicomputers", *Concurrency: Practice and Experience*, **5**(5), pp. 407–423, Aug. 1993.

[2] M.I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge, MA, 1989.

[3] H. Zima and B. Chapman, "Compiling for Distributed-Memory Systems", Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines, Feb. 1993.

[4] T.J. Biggerstaff, "The Concept Assignment Problem in Program Understanding", Procs. *IEEE Working Conf. on Reverse Engineering*, May 21-23, Baltimore, Maryland, USA, 1993.

[5] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis", *IEEE Software*, Jan. 1990.

[6] L.M. Wills, "Automated Program Recognition: a Feasibility Demonstration", *Artificial Intelligence*, **45**, 1990.

[7] W. Kozaczynski, J. Ning and A. Engberts, "Program concept Recognition and transformation", *IEEE Trans. on Software Engineering*, Dec. 1992.

[8] I. Foster and K. M. Chandy, "Fortran M: A language for modular parallel programming", *J. Parallel Distributed Systems*, 1994.

[9] B. Chapman, P. Mehrotra, J. Van Rosendale and H. Zima, "A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism", Proc. *CONPAR'94 – VAPP VI*, pp.664-676, Linz, Austria, Sep. 1994.

[10] B. Di Martino and G. Iannello, "Towards Automatic Parallelization through Program Comprehension", Procs. 3^{dh} *IEEE Workshop on Program Comprehension*, Washington (USA) - Nov. 1994.

[11] B. Di Martino, "PAP Recognizer: a Pattern Recognition Tool aimed to Code Parallelization", Tech. Rep. Institute for Software Technology and Parallel Systems - Univ. of Vienna, Apr. 1995 (to appear).

[12] I. Foster, C. Kesselman, R. Olson and S. Tuecke, "Nexus: An interoperability layer for parallel and distributed computer systems", Tech. Rep. ver. 1.3, Argonne National Labs, Dec. 1993.

[13] M. Haines, D. Cronk and P. Mehrotra, "On the design of Chant: A talking threads package", ICASE Report 94-25, Institute fo Computer Applications in Science and Engineering, NASA Langley Research Center - Hampton VA, Apr. 1994.

[14] J. Li and M. Chen, "Index Domain alignment: minimizing cost of cross-referencing between distributed arrays", Tech. Rep. Dep. of Computer Science, Yale University, n. 725, Nov. 1989.

[15] M. Gupta, "Automatic Data Partitioning on distributed memory computers", Diss., University of Illinois at Urbana-Champaign, 1992.

[16] T. Fahringer, "The Weight Finder, An Advanced Profiler for Fortran Programs", in *Automatic Parallelization, New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Verlag-Vieweg, pp. 1-24, Mar. 1993.

[17] G.C. Fox et al., *Solving Problems on Concurrent Processors*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[18] R. Aversa, "Caratteristiche dell' ambiente di programmazione parallela MPI", Int. Rep. Dept. of Computer Science and Systems, Univ. "Federico II" - Naples, 1995 (to appear).

[19] C. W. Kessler and W. J. Paul, "Automatic Parallelization by Pattern Matching", *Lecture Notes in Computer Science*, (734) Parallel Computation, 1993.

[20] S. Bhansali, J. R. Hagemeister, C. S. Raghavendra and H. Sivaraman, "Parallelizing sequential programs by Algorithm-level Transformations", Procs. 3^{dh} *IEEE Workshop on Program Comprehension*, Washington (USA) - Nov. 1994.

ADDT: Automatic Data Distribution Tool for Porting Programs to Distributed Memory Machines

Harikumar Sivaraman and Cauligis Raghavendra
Washington State University

This paper presents a method for automatically generating a data distribution for a distributed memory machine. This method has been implemented in an automatic data distribution tool called ADDT. ADDT reads in a shared memory parallel program in Fortran and generates as output a HPF program.

ADDT works in three steps. In the first step it divides the input program into sub parts called distribution blocks (DB). (A somewhat similar idea has been suggested by other researchers). In the second step it obtains a data distribution for the arrays in a given block by trying to use the data distribution from preceding blocks. If this fails it generates a new distribution for the given block. In the third step ADDT generates redistribution statements for those arrays that have different distributions in different parts of the program. These redistribution statements are generated as early in the program as possible so as to allow a HPF compiler to overlay communication with computation.

To find a distribution for a given DB, ADDT first constructs a list of array references in the DB. Then for each pair of array references it represents the difference in their subscript expressions using a data access pattern descriptor. ADDT then does a table lookup using these data access pattern descriptors to obtain the data distribution for the given DB. For each block ADDT tries to find a data distribution that is totally communication free. If it can't find such a distribution it then chooses a distribution that minimizes the number of elements to be communicated. ADDT also performs detailed inter-procedural analysis and it supports procedure cloning.

ADDT: Automatic Data Distribution Tool for Porting Programs to Distributed Memory Machines *

H. Sivaraman and C.S. Raghavendra
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164 - 2752
{ *hsivaram, raghu* } *@eecs.wsu.edu*
Ph: (509) 335 - 8246, Fax: (509) 335 - 3818

Abstract

Data parallel languages like HPF require the user to specify the distribution of the data in the application. This distribution is specified over the processors on which the application is to execute. A good data distribution depends on the number of processors, the communication latency and the underlying compilation system. Hence it is difficult to do this manually.

In this paper we present a method for automatically generating a data distribution for a distributed memory environment. This method has been implemented in an *automatic data distribution tool* called ADDT. ADDT reads in a shared memory parallel program in Fortran and generates as output a HPF program. Our method works by breaking up the input program into sub parts called *distribution blocks* (**DB**). (A somewhat similar idea has been suggested by other researchers). It then obtains a data distribution for the arrays in each DB using the data layouts from preceding DB's. The

*This research is supported by the Boeing Centennial Chair Professor Funds.

algorithms used to obtain a data distribution for a DB are also discussed in this paper. We present experimental results obtained by running ADDT on some example programs.

1 Introduction

Distributed memory machines are a promising means of achieving high performance computing for scientific applications. These machines offer an advantage over shared memory machines in that they are scalable and are more cost effective, but are more tedious to program since they have a fragmented memory space. Any program written for these machines has to contain specifications for the distribution of the data in the program over the memories that are attached to the different processors. Data parallel languages like HPF provide the user with constructs [20]* to specify the data layout of the arrays in the application.

The manner in which the data is distributed across the different memories significantly affects the performance of the program. Therefore it is important to choose a distribution that is good on the particular architecture on which the program is to run. A good data distribution depends on the number of processors, the communication latency of the environment, and the underlying compilation system. Hence it is difficult for a user to obtain a good data distribution for a given program.

The objective of this paper is to present that part of a source to source compiler that performs an automatic data distribution. Our algorithms have been implemented in an automatic data distribution tool called ADDT. ADDT reads as input a parallel program written for a shared memory machine and generates as output a HPF program. ADDT generates a data distribution for multidimensional arrays. It performs detailed inter-procedural analysis [10] and it supports procedure cloning [10].

ADDT works in three steps. In the first step it divides the input program into sub parts called *distribution blocks* (**DB**). (A somewhat similar idea has been suggested by other researchers [14] [12]†). In the second step it obtains a data distribution for the arrays in a given block by trying to use the data distribution from preceding blocks. (We try to

*may be obtained at cs.rice.edu/public/HPFF

†these technical reports may be obtained at http://www.cs.rice.edu/~kremer

use the distributions from preceding blocks so that we can reduce data redistribution to a considerable extent).If this fails it generates a new distribution for the given block. In the third step ADDT generates *redistribution* statements for those arrays that have different distributions in different parts of the program. These redistribution statements are generated as early in the program as possible so as to allow a HPF compiler to overlay communication with computation.

To find a distribution for a given DB, ADDT first constructs a list of array references in the DB. Then for each pair of array references it represents the difference in their subscript expressions using a *data access pattern descriptor*. ADDT then does a *table lookup* using these data access pattern descriptors to obtain the data distribution for the given DB. For each block ADDT tries to find a data distribution that is totally communication free. If it can't find such a distribution it then chooses a distribution that minimizes the number of elements to be communicated.

In section 2 we discuss the internal representation in ADDT of the data layout of an array. In section 3 we present the algorithms used by ADDT to obtain a data distribution for a DB. We illustrate the algorithms in section 3 with two simple examples in section 4. We present experimental results in the section 5. In this paper we do not discuss the method used by ADDT to do *redistribution*, nor do we discuss the *inter-procedural analysis* methods used by ADDT.

2 Internal representation of data distribution

Data parallel languages like HPF provide constructs [20] to express the data layout of the arrays in an application. These constructs require the specification of the distribution of each dimension of the array using one of the three primitives, *BLOCK*, *CYCLIC*, and $\star$. The constructs used by ADDT to represent internally the data distribution of an array differ considerably from those used by HPF. In the final phase of data distribution ADDT converts its internal representation to *HPF* format so as to comply with the standard. We chose to use this different representation because we believe that it is easier to work with while trying to obtain a data distribution for a program. In this section we shall discuss the internal representation used by ADDT.

The internal representation is characterized by two parameters:

a.) Block size.

b.) Repetition vector.

2.1 Block size

When an array is distributed each processor holds one or more contiguous elements of the original array. The number of contiguous elements owned by each processor is the *block size*. The smallest size of a block is one (such a distribution is often called a *CYCLIC* distribution by some researchers) and its largest size is the size of the array itself. Our compiler tries to obtain a block size between these two extreme values for the arrays in the program.

Example 2.1

```
doall i = 1, nx-3
      a(i) = b(i)
enddo
```

In this case the compiler will assign a block size so as to keep all processors active. So it assigns to array *'a'* and to array *'b'* a *block size* of $\lceil \frac{nx}{p} \rceil$, where p is the number of processors. In memory, an array is always a single dimensional construct even if it is declared as a multidimensional structure. Hence the parameter *block size* is a natural way to capture the data layout of an array. It is important to note that though the internal representation treats an array as a unidimensional construct ADDT does not strip off or ignore the dimension information available in the source program.

2.2 Repetition vector

The second parameter, the repetition vector is explained using example 2.2.

Example 2.2

```
integer a(5, 10, 10), b(5, 10, 10)
doall i = 1, n
      doall j = 1, m
```

```
            doall k = 1, m-1
            a(i, j, k) = b(i, j, k) + b(i, j+1, k+2)
            enddo
        enddo
    enddo
```

In the example 2.2, in the initial distribution step the compiler picks a block size of **1** for array *'a'*if *'p'*is a divisor of 5. Hence a processor p_l owns elements a(p_l+1, j, k), b(p_l+1, j, k) and b(p_l+1, j+1, k) $\forall$ j, k. This ensures that all the data items that each processor needs to execute its share of this loop are available locally. But if p $>$ 5 this data distribution makes it necessary for processors to exchange elements of array *'b'*. ADDT eliminates the need to communicate by modifying the data distribution suggested above. This modification consists of the repetition vector.

Let *'p'*the number of processors be 10. Then if the compiler picks a block size of 1 the repetition vector is set to (5, 100). The data distribution for array *'a'*is as shown in Table I. In Table I each box has two numbers, the first number is the index of the element in the array *'a'*, and the second number is the number of the processor which owns that element. In this table only the first 100 elements of array *'a'*have been shown. There are four more such blocks of 100 elements and each of these four blocks is distributed in an identical manner. The first element of the repetition vector says how many elements are distributed without repetition. In example 2.2 this element is 5, this means that five elements are distributed with the block size that was chosen. Let us call this distribution the *primary distribution*. In this example this primary distribution is repeated over 50 elements because each iteration accesses successive columns of the array *'b'*. The 51^{st} element is assigned to processor 5. Starting from the 51^{st} element the repetition vector is used to assign the elements of array *'a'*to the processors. The resultant distribution is shown in Table I. The second element of the repetition vector says after how many elements the distribution has to be repeated. In this example this element is 100. Hence the distribution is repeated after 100 elements. This is because each iteration accesses alternate elements from the third dimension of array *'b'*. In general, the repetition vector is an *(n-1)-tuple*, the first element specifies the size of the primary distribution, the second and subsequent elements specify the number of elements **after** which the distribution pattern is repeated.

1 : 0	6 : 0	11 : 0	16 : 0	21 : 0	26 : 0	31 : 0	36 : 0
2 : 1	7 : 1	12 : 1	17 : 1	22 : 1	27 : 1	32 : 1	37 : 1
3 : 2	8 : 2	13 : 2	18 : 2	23 : 2	28 : 2	33 : 2	38 : 2
4 : 3	9 : 3	14 : 3	19 : 3	24 : 3	29 : 3	34 : 3	39 : 3
5 : 4	10 : 4	15 : 4	20 : 4	25 : 4	30 : 4	35 : 4	40 : 4
41 : 0	46 : 0	51 : 5	56 : 5	61 : 5	66 : 5	71 : 5	76 : 5
42 : 1	47 : 1	52 : 6	57 : 6	62 : 6	67 : 6	72 : 6	77 : 6
43 : 2	48 : 2	53 : 7	58 : 7	63 : 7	68 : 7	73 : 7	78 : 7
44 : 3	49 : 3	54 : 8	59 : 8	64 : 8	69 : 8	74 : 8	79 : 8
45 : 4	50 : 4	55 : 9	60 : 9	65 : 9	70 : 9	75 : 9	80 : 9
81 : 5	86 : 5	91 : 5	96 : 5				
82 : 6	87 : 6	92 : 6	97 : 6				
83 : 7	88 : 7	93 : 7	98 : 7	...			
84 : 8	89 : 8	94 : 8	99 : 8				
85 : 9	90 : 9	95 : 9	100 : 9				

Table I: Data distribution with a block size of 1 and a repetition vector of (5, 100).

ADDT uses these two parameters to specify the data distribution of an array in a program. Every array that is distributed[‡] has a block size attribute but only some arrays need a *repetition vector* to describe their distribution.

3 Data distribution for a distribution block

A *distribution block*(DB) is defined as one of:

- A sequence of statements with no loops or call statements or function calls or *DO* or *DOALL* loops. Such a sequence is henceforth called a **straight sequence** of statements.
- A *straight sequence* of statements between a *DO* or *DOALL* loop and the next *DO* or *DOALL* loop.

[‡]the user can specify at compile time the minimum size of an array that may be distributed. The default minimum size is one. This prevents ADDT from distributing small arrays.

Algorithm Table_lookup

Input: Lists **'DD'**, **'S'**, **'D'**

Output: List of data distributions, DD_1

Foreach element in **'S'** which is of the form (A_i^k, A_j^m)

use element d_i from the list **'D'** to do a table lookup

if the table lookup is successful the result will be a data distribution for the array A^k

scale this distribution to fit the start and end elements of

the array A^k and insert in the list DD_1

endfor

Figure 1: Algorithm for table lookup

- A *straight sequence* of statements between the most deeply nested *DO* or *DOALL* loop and its corresponding *ENDDO* or *CONTINUE* statement.
- A call statement or statement containing a function call.

To obtain a distribution for a given distribution block ADDT uses the algorithm in figure 3. For each DB ADDT tries to obtain a data distribution that is totally communication free. If this is not possible ADDT finds[§] a data distribution that minimizes communication.

For each DB ADDT first constructs a list **'L'** of array references in the DB. It then constructs another list **'S'** which consists of pairs of elements from the list **'L'**. The list **'S'** can contain at most $n * (n-1)$ elements, where n = | 'L' |. In most cases the list **'S'** will have fewer elements. (see the algorithm in figure 2 for details). Each element in the list **'S'** is a pair of array references. For each element in **'S'** ADDT represents the difference in the subscripts using a *data access pattern descriptor*. ADDT then does a *table lookup* using these data access pattern descriptors to obtain the data distribution for the arrays in the given DB.

[§]provided the user asks ADDT to do so by setting the switch '-x'.

Algorithm Dist_block

Input: A distribution block

Output: The lists **'D'**, **'S'**, **'L'**

construct a list of the array references in the DB.

let $\mathbf{L} = (A_1^1, A_2^1, \ldots, A_n^k)$ where A_i^k is a reference to array A^k.

Construct a list $\mathbf{S} = (A_i^k, A_j^m)$, $\forall\ A_i^k, A_j^m \in$ L

The set $\mathbf{S}$ can contain at most $n * (n-1)$ elements where n $= |L|$

From $\mathbf{S}$ delete those elements (A_j^k, A_i^m) if $(A_i^k, A_j^m) \in \mathbf{S}$.

Foreach pair $(A_i^k, A_j^m) \in \mathbf{S}$ where $A_i^k = A^k(i_1, i_2, \ldots, i_x)$, $A_j^m = A^m(i'_1, i'_2, \ldots, i'_y)$

define $D_{ij}(d_1, d_2, \ldots, d_l)$, l = max(x, y)

where d_t, $\forall$ t such that $1 \leq t \leq l$ is:

if $(i_t$ - $i'_t) = 0$ and i_t, i'_t are affine expressions of a loop index then

$d_t =$ '0'

else if $(i_t$ - $i'_t) = 0$ and i'_t are NOT affine expressions of a loop index then

$d_t =$ '0'

else if i_t - i'_t is an integer and i_t, i'_t are affine expressions of a loop index then

$d_t =$ 'C'

else if i_t - i'_t is an integer and i_t, i'_t are NOT affine expressions of a loop index then

$d_t =$ 'c'

else $d_t =$ 'E'

put D_{ij} into the list $\mathbf{D}$

endfor

Figure 2: Algorithm to obtain a data distribution for a DB.

Algorithm Data_dist

Output: List of data distributions, **'DD'**

Foreach DB

pick up the data distributions of all the arrays in the DB

Put these data distributions in a list **'DD'**

(These distributions were assigned to the arrays from preceding DB's)

If an array has no distribution assigned to it yet set its *block size* to -1

Invoke Algorithm Dist_block

Line α : Invoke Algorithm table lookup on the lists **'D'** and **'S'**

with the data distributions in list **'DD'**

if Algorithm table lookup fails set list **'DD'** to a null list

Invoke Algorithm table lookup on the lists **'D'** and **'S'**

if Algorithm table lookup fails once again

DO

delete all occurrences of an array(exactly which array is

deleted is a matter of implementation detail) from lists **'D'** and **'S'**

put the deleted array in the list **'P'**.

Invoke Algorithm table lookup on the lists **'D'** and **'S'**

Line β : while (**'S'** is not empty or a distribution is not yet found)

put the distributions that were obtained into the list DD'

Reverse the lists, **'L'**, **'D'**, and **'S'**

Repeat the lines between Line α and Line β

put the distributions that were obtained into the list DD''

put those elements in lists DD', and DD'' that are identical into list **'DD1'**

if DD' is longer than DD''

reverse lists **'S'**, **'L'**, and **'D'**

repeat the steps between Line α and Line β

use the obtained distributions to assign data distributions for the arrays in the list **'P'**

append the distributions obtained to the list **'DD'**

endfor

Figure 3: Algorithm to obtain a data distribution

4 Examples

In this section we use a simple example to show how ADDT uses the algorithms presented in the previous section. In practice ADDT handles much harder programs.

Let the number of processors be p = 10.

Example 4.1

```
integer a(5, 10, 10), b(5, 10, 10)
doall i = 1, n
    doall j = 1, m
        doall k = 1, m-1
        a(i, j, k) = b(i, j, k) + b(i, j+1, k+2)
        enddo
    enddo
enddo
```

For example 4.1 ADDT constructs the list **'L'**; **'L'** = (b(i, j, k), b(i, j+1, k+2), a(i, j, k)). For the rest of the discussion of this example we'll represent **'L'** as **'L'** = (A_1^2, A_2^2, A_1^1)[¶] The list **'S'** = ($\{A_1^2, A_2^2\}$, $\{A_1^1, A_1^2\}$, $\{A_1^2, A_1^1\}$ $\{A_1^1, A_2^2\}$, $\{A_2^2, A_1^1\}$) The list **'D'** is (D_{12}(OCC), D_{11}(OOO), D_{11}(OOO), D_{12}(OCC), D_{21}(OCC)).

The *access pattern descriptors* are OCC, OOO, OOO, OCC, OCC. The algorithm table lookup takes the first descriptor OCC, and the corresponding elements of the lists **'S'**, **'L'**, and **'D'**. This descriptor has three elements. It looks up in a table for each element of this descriptor[‖]. After looking up the first element of the first descriptor algorithm table lookup assigns to array A^2 a block size ≥ 1(this means that array A^2 can have any value of block size ≥ 1). It then picks up the next element of the descriptor OCC, looks up the table and sets the block-size of array A^2 to 50 which is the size of the first two dimensions. It then picks

[¶] **'L'**, **'S'**, **'D'** are lists not sets and the order of the elements in the lists is important and an element can be repeated.

[‖] The contents of the table are presented in Appendix A

up the last element of the descriptor OCC, looks up the table but now the only matching entry has a block size of 500 which is the size of the first three dimensions. This block size is unacceptable because it will keep only one processor busy and there are 10 processors. At this stage the algorithm table lookup goes back to the second element of the descriptor, OCC and runs it through the table. This time the matching entry has a block size of 1 and a repetition vector of (5). The algorithm table lookup then picks up the last element of the descriptor, OCC, looks up the table and finds a match with a block size of 1 and an extension of the repetition vector obtained earlier. The repetition vector is now (5, 100). ADDT repeats this process for the other elements in the list **'D'** and finally obtains the data distribution for array **'a'** and **'b'**. This distribution has a block size of 1 and a repetition vector of (5, 100).

This algorithm is in the worst case exponential in the number of dimensions of an array. But since programs rarely have arrays with more than 7 or 8 dimensions the number of table lookups is at most 2^8.

5 Experimental Results

The data distributions obtained by running ADDT on some example programs is shown in table II. All of these programs are sequential FORTRAN programs. The sequential programs were first parallelized using Parafrase-2. The output from Parafrase-2 is a parallel program for shared memory machines. This parallel program was input to ADDT. The results of the analysis performed by ADDT are shown in Table II. The table also has distributions obtained by a programmer for some of the programs. For the other programs we use the distributions obtained by other researchers.

6 Conclusions and future work

In this paper we have presented the algorithms used by ADDT (an automatic data distribution tool) to generate automatic data distributions of arrays in source programs. ADDT

Program	Data decompositions obtained by ADDT	Decompositions obtained by a programmer	Decomposition obtained by other researchers
Jacobi[1] (1Kx1K)	A(BLOCK, *), B(BLOCK, *)	A(BLOCK, *), B(BLOCK, *)	$A(*, BLOCK)$[3] $B(*, BLOCK)$[3]
sor[1]	A(BLOCK, *)	A(BLOCK, *)	$A(BLOCK, *)$[3]
ADI[1]	U1(BLOCK, BLOCK, *,) U2(BLOCK, BLOCK, *,) U3(BLOCK, BLOCK, *,) DU1(BLOCK,) DU2(BLOCK,) DU3(BLOCK,)		$U1(BLOCK, BLOCK, *,)$[3] $U2(BLOCK, BLOCK, *,)$[3] $U3(BLOCK, BLOCK, *,)$[3] $DU1(BLOCK,)$[3] $DU2(BLOCK,)$[3] $DU3(BLOCK,)$[3]
Matrix Multiplication $C_{nXn} = A_{nXn} * B_{nXn}$	C(BLOCK, *) A(BLOCK, *) B(*, *)	C(BLOCK, *) A(BLOCK, *) B(*, *)	
Gaussian Elim A_{nXn} →upper triangular form	A(BLOCK, *)	A(BLOCK, *)	
old.drn.f[2]	X(BLOCK), Y(BLOCK) Z(BLOCK), FX(BLOCK) FY(BLOCK), FZ(BLOCK) Energ(BLOCK), Ney(BLOCK, *)	X(BLOCK) Y(BLOCK) Z(BLOCK) FX(BLOCK) FY(BLOCK) FZ(BLOCK) Energ(BLOCK) Ney(BLOCK, *)	
erlebacher[1]	dx3d6p : DUX(*, BLOCK, *,) tridvpi : DUX(BLOCK, *, *) DUX(*, BLOCK *,) DUY(BLOCK, *, *) DUZ(BLOCK, *, *)		$DUX(*, *, BLOCK)$[4] $DUY(*, *, BLOCK)$[4] $DUZ(*, BLOCK, *)$[4]

[1] These programs are from the set of examples in the **DSystem**

[2] This is an 800 line program that computes the steady state of a system of atoms. This program was developed by Dr. R. Hoagland in the Dept. of material science, WSU.

[3] Taken from the examples in the **DSystem**

[4] From the SUIF project.

Table II: Data distributions obtained by ADDT

reads as input a parallel program written for shared memory machines. It generates as output the corresponding HPF program. To obtain a data distribution for a given DB, ADDT breaks up a program into sub-parts called distribution blocks. To find a distribution for a DB, ADDT generates a list of array references, a corresponding list of data access pattern descriptors, and a list of data distributions from preceding distribution blocks. It then uses these lists to do a table lookup to obtain the final data distribution . Whenever the distribution of an array changes ADDT generates redistribution statements. ADDT performs detailed inter-procedural analysis. It supports procedure cloning.

It has been our experience that the data distribution that is obtained changes significantly based on the communication speed of the distributed memory environment. For example, on a network of workstations with PVM, it is often good to replicate all arrays that are only read in a loop section whereas on a machine like the Intel Paragon this may not be necessary to achieve a speedup. ADDT does not perform these target specific optimizations. To find a data distribution for a DB our methods use the distribution obtained from preceding blocks. We do not try to use the distribution from the following block. i.e. we do not have a *lookahead*. We are working to find a lookahead mechanism other than backtracking(Backtracking is too expensive computationally).

Acknowledgement

The authors would like to thank Jack R. Hagemeister for building the GUI.

References

[1] Jennifer M Anderson, Saman P. Amarasinghe, and Monica Lam. Data and computation transformations for multiprocessors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995. Santa Barbara, CA.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90d/hpf compiler for distributed memory mimd computers: Design, implementation, and performance results. In *Supercomputing*, November 1993. Portland, OR.

[3] T. Brandes. *ADAPTOR: Language reference manual.* German Nat'l Research Centre for Computer Science, version 2.0 edition, March 1994.

[4] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimising performance. *Journal of Supercomputing*, pages 171–207, October 1992.

[5] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and C.-W Tseng. Compiling fortran 77d and 90d for mimd distributed memory machines. In *Symposium on the Frontiers of Massively Parallel Computation*, October 1992. McLean, Virginia.

[6] P Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *Int'l Conf. on Supercomputing*, July 1993. Japan.

[7] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Distributed Memory Computing Conf.*, April 1991. Portland, Or.

[8] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 3, March 1992.

[9] M. Gupta and P. Banerjee. Paradigm: A compiler for automatic data distribution on multicomputers. In *Int'l Conf. on Supercomputing*, July 1993. Japan.

[10] M. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of fortran d for mimd distributed-memory machines. In *Supercomputing*, November 1992. Minneapolis.

[11] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for fortran d on mimd distributed-memory machines. In *ACM Intl. Conf. on Supercomputing*, July 1992. Washington, DC.

[12] K. Kennedy and U. Kremer. Automatic data layout for high performance fortran. Technical Report CRPC-TR94498-S, Centre for Research on Parallel Computation, Rice Univ., December 1994.

[13] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distributed Systems*, October 1991.

[14] U. Kremer, John Mellor-Crummey, K Kennedy, and Carle Alan. Automatic data layout for distributed-memory machines in the d programming environment. Technical Report CRPC-TR93298-S, Centre for Research on Parallel Computation, Rice Univ., February 1993.

[15] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, pages 213–221, October 1991.

[16] L. M. Liebrock and K. Kennedy. Parallelization of linearized applications in fortran d. In *Intl. Conf. on Parallel Processing*, April 1994. Cancún, Mexico.

[17] Ravi Mirchandaney, Seema Hiranandani, and Ajay Sethi. Improving the performance of dsm systems via compiler involvement. In *Supercomputing*, November 1994.

[18] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A portable 'shared-memory' programming model for distributed memory computers. In *Supercomputing*, November 1994.

[19] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee. Communication optimizations used in the paradigm compiler for distributed-memory multicomputers. In *Intl. Conf. on Parallel Processing*, August 1994.

[20] High performance fortran forum. *High performance fortran language specification*, version 1.0 edition, May 1993.

[21] C. S. Raghavendra and V. K. Prasanna Kumar. Permutations on illiac iv-type networks. *IEEE Transactions on Computers*, pages 662–669, July 1986.

[22] J. Ramanujam and P. Sadayappan. Compile time techniques for data distribution in distributed memory machines. *IEEE Trans. on Parallel and Distributed Systems*, pages 472–481, October 1991.

[23] M. Rosing, R. B. Schnabel, and R. P. Weaver. The dino parallel programming language. *Journal of Parallel and Distributed Computing*, pages 30 – 42, 1991.

[24] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic mimd/simd parallelization. *Parallel Comput.*, 6:1–8, 1988.

Appendix A

The contents of the table which is used by ADDT to obtain a data distribution for are listed in this section** . The table has been implemented as a sequence of *if then else* statements.

The table lookup algorithm in figure 1 works with a descriptor, a pair of array references, (A_i^k, A_j^l). and a list of data distributions obtained so far, **DD**. On completing the table lookup it returns a data distribution for the array A^k. If it could not find a data distribution for the array A^k given the descriptor it returns a *failure*.

Definitions:

Let array A^k have **s** dimensions and be declared as $A^k(u_1, u_2, \ldots, u_s)$ where u_1 is the size of the first dimension, u_2 that of the second dimension and so on.

maximum load : It is defined as the total size of an array divided by the number of processors. For the array A^k *maximum load* would be $\frac{(\prod_{i=1}^{s} u_i)}{p}$

BEGIN TABLE

if (block size undefined **and** descriptor element is one of 'c' or '0'
and $u_1 \leq$ the maximum load given that there are *'p'*processors)
block size is set to u_1
if (block size undefined **and** (descriptor element is 'O' or if user
has chosen a '-x' option))
if the corresponding array subscripts refer to parallel loop indices
block size is set to anything ≥ 1
if the corresponding array subscripts refer to sequential loop indices
and $u_1 \leq$ the maximum load given that there are *'p'*processors
block size is set to u_1
if the corresponding array subscripts refer to sequential loop indices

**the table as implemented in ADDT is more detailed but resembles the description in this section in essence

and a data distribution needs to be **forced**

(the condition **forced** is decided based on the flag '-x'

and the number of elements that need to be communicated as a consequence

of distribution. This decision process is not discussed in the paper.)

block size is set to anything ≥ 1

if (block size undefined **and** descriptor element is 'C')

if the corresponding array subscripts refer to parallel loop indices **and** the subscripts are ide

block size is set to anything ≥ 1

if the corresponding array subscripts refer to sequential loop indices

block size is set to u_1

if the corresponding array subscripts refer to parallel loop indices **and** the

(difference in subscripts)*$u_1 \leq$ the maximum load given that there are *'p'*processors)

block size is set to u_1

if the corresponding array subscripts refer to parallel loop indices **and** the

(difference in subscripts)*$u_1 >$ the maximum load given that there are *'p'*processors)

and a data distribution needs to be **forced**

block size is set to anything ≥ 1

if (block size undefined **and** descriptor element is 'E')

if the difference in subscripts)*$u_1 \leq$ the maximum load given that there are *'p'*processors

block size is set to u_1

if difference in subscripts*$u_1 >$ the maximum load given that there are *'p'*processors

and a data distribution needs to be **forced**

block size is set to anything ≥ 1

if (block size defined **and** (descriptor element is 'O' or a data distribution

needs to be **forced**) **and** corresponding loop indices are parallel

leave the block size unchanged.

if (block size defined **and** descriptor element is one of 'C', 'c', '0', 'E' **and**)

the corresponding loop type is parallel

if the current value of block size is *leq* the maximum load given that there are *'p'* processors
and difference in subscripts is zero
leave the block size unchanged.
if the current value of block size is > the maximum load given that there are *'p'* processors
and difference in subscripts is zero
reset the block size and create a *repetition vector*
if difference in subscripts*$u_1 \leq$ the maximum load given that there are *'p'* processors
leave the block size unchanged.
if difference in subscripts*$u_1 >$ the maximum load given that there are *'p'* processors
reset the block size and create a *repetition vector*

if (block size defined **and** descriptor element is one of 'C', 'c', '0', 'E' **and**)
the corresponding loop type is sequential
if the current value of block size is *leq* the maximum load given that there are *'p'* processors
leave the block size unchanged.
if the current value of block size is > the maximum load given that there are *'p'* processors
reset the block size and create a *repetition vector*

END TABLE

Automatic Mapping of Task and Data Parallel Programs

Jaspal Subhlok, Carnegie Mellon University

Many applications in a variety of domains including digital signal processing, image processing, and computer vision are composed of a sequence of tasks that act on a stream of input data sets in a pipelined manner. These applications are best mapped to a massively parallel machine by dividing the tasks into modules and assigning a subset of the available processors to each module. Recent research has established the performance advantage of "combining task and data parallelism" over pure data parallel and task parallel approaches. We have demonstrated that realistic programs can execute several times faster if they use a good mapping instead of a simple data parallel mapping.

This talk addresses the problem of optimally mapping task and data parallel programs onto a massively parallel machine. For a program that acts on a stream of inputs, there are several possible optimization goals. One may wish to maximize the throughput, which is the number of data sets processed per second, or to minimize the latency, which is the elapsed time between reading an input data set and producing the corresponding output data set. In general, the goal is to optimize for throughput, latency, or number of processors, possibly with constraints on the others. We formulate the general optimization problem for data parallel pipelines and present a class of optimal solution algorithms.

Our formulation uses a general and realistic model for inter-task communication, takes memory constraints into account, and addresses the entire problem of mapping which includes clustering tasks into modules, assignment of processors to modules, and possible replication of modules. The optimal algorithms are based on dynamic programming and the execution complexities are low polynomials in terms of the number of available processors and the number of tasks. We also present linear heuristic algorithms that can be used to solve the problem efficiently in practice. The entire framework is integrated with the Fx parallelizing compiler for High Performance Fortran and is part of a tool that automatically maps task and data parallel programs onto massively parallel machines. We present experimental results that demonstrate the importance of choosing a good mapping and show that the methods presented yield efficient mappings and predict optimal performance accurately.

P^3T: An Automatic Performance Estimator for Parallel Programs

Thomas Fahringer, University of Vienna, Austria

In this talk I will discuss the P3T, which is a static and automatic performance estimator for data parallel programs. It computes at compile time a set of parallel program parameters that characterize the behavior of regular parallel programs. This includes work distribution, number of data transfers, amount of data transferred, transfer times, network contention, and number of cache misses. These parameters can be selectively obtained for statements, loops, procedures, and the entire program.

The P^3T is based on an analytical performance model, whose computational complexity is independent of the program's problem size, statement execution and loop iteration counts. As a consequence, estimating the parallel program parameters is considerably faster than simulating or actually executing the program.

A graphical user interface relates performance data back to the original parallel code, enables to immediately identify performance hot spots, and allows to filter and display performance data at various levels of detail.

Experiments show the ability of the P^3T to successfully guide both programmer and compiler in the effort of parallelizing and optimizing programs. The results are very encouraging, and demonstrate the feasibility of automatic performance prediction for parallel programs.

P^{3}T: An Automatic Performance Estimator for Parallel Programs

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1092 Vienna, Austria
e-mail: tf@par.univie.ac.at

Abstract

The area of parallelizing compilers for distributed memory multicomputers has seen considerable research activity during the last few years. Most of the current compilers do not provide any support for estimating performance impacts of code changes that they apply. In this paper, we present P^3T, which is a static and automatic performance estimator for data parallel programs. It computes at compile time a set of parallel program parameters that characterize the behavior of the parallel program. This includes work distribution, number of data transfers, amount of data transferred, transfer times, network contention, and number of cache misses. These parameters can be selectively obtained for statements, loops, procedures, and the entire program; furthermore, their effect with respect to individual processors can be examined. Based on these parameters the programmer can apply well-directed program changes to eliminate or alleviate every specific performance drawback as indicated by the parallel program parameters.

P^3T is based on an analytical performance model, whose computational complexity is independent of the program's problem size, statement execution and loop iteration counts. As a consequence, estimating the parallel program parameters is considerably faster than simulating or actually executing the program.

A graphical user interface relates performance data back to the original parallel code, enables identifying of performance hot spots, and allows filtering and displaying of performance data at various levels of detail.

Experiments show the ability of P^3T to successfully support both programmer and compiler in the effort of parallelizing and optimizing programs. The results are very encouraging, and demonstrate the feasibility of automatic performance prediction for parallel programs.

1 Introduction

Scalable high performance computing is an attractive means for meeting the ever increasing demand for greater computing power. Distributed memory machines offer significant advantages over their shared memory counterparts in terms of cost and scalability. However, there has not yet been a massive breakthrough of distributed memory parallel processing due to the tedious, error-prone and time consuming efforts to program the underlying parallel architectures. State-of-the-art parallelizing compilers support a global address space, numerous program transformations and data distribution strategies to simplify this task. Nevertheless, most of the current compilers leave the responsibility to make most of the strategic decisions which includes the application of program transformations and the search for a good data distribution scheme almost entirely to the user. Rather than focusing on the development of parallel algorithms, the programmer spends a significant amount of time studying architecture and compiler-specific details. Obviously, a performance estimator would strongly support both programmer and compiler in carefully evaluating performance gains and trade-offs among applicable program restructuring techniques.

The area of performance prediction for parallel programs has seen considerable research activity during the last few years: V. Sarkar ([29]) estimates the runtime of parallel programs at compile time incorporating profiling and pre-measured kernel codes. This approach targets single assignment languages, which alleviates analysis significantly. Balasundaram et. al. ([1]) introduced a performance estimator to evaluate different data partitioning strategies by using pre-measured kernel codes. Their approach achieves good accuracy for the loosely synchronous programming

model. K.Y. Wang ([31]) characterizes parallel programs by a parameterized performance model, which aims at runtime information. The cache and network contention behavior is considered only very superficially. A recent update of this work ([24]) discusses a more advanced estimation approach using symbolic analysis. A. van Gemund ([17]) designed the Pamela performance modeling methodology, which provides a theoretical framework for modeling and analyzing serialization effects and the performance of parallel systems. Although petri nets ([2, 15]), queueing networks ([22]) and markov chains ([30]) can be valuable in understanding the dynamic behavior of parallel programs, the associated analysis costs both in terms of runtime and memory requirements prohibits their use in compilers. N. Yazici-Pekergin and J.M. Vincent ([33]) obtain stochastic bounds on execution times of parallel programs assuming the availability of an unlimited number of processors. The execution times of parallel tasks are independent identically distribution random variables. F. Hartleb and V. Mertsiotakis ([20]) derive upper and lower execution time bounds for parallel programs with the aim to tune mapping and implementation alternatives. Parallel programs are modeled as a stochastic graph and a random variable describes the runtime behavior of a specific processor. J. Prost and S. Kipnis ([27]) describe a multi-level trace-driven simulation method to analyze the performance of programs for distributed memory parallel systems. This approach is based on an execution trace that consists of a sequence of program events to be simulated. Its architecture model does not account for some critical components such as data memory including caches. Most of these approaches have serious drawbacks associated with them. Some of them have a problem of restricted applicability, they apply only to programs that may be modeled as single, multiple nested loops with constant loop bounds and no procedure calls. Some others are too expensive in terms of runtime and memory requirements. Purely theoretic approaches are difficult to be validated against actual program behavior on real architectures. Frequently guessing is applied to obtain concrete values for program unknowns such as loop bounds. User-interfaces commonly lack performance data filtering and rarely relate performance data back to the original source code.

This paper describes P^3T – a *Parameter based Performance Prediction Tool* – which is a performance estimator for data parallel programs based on distributed memory parallel architectures. The goal of P^3T is to provide the programmer and parallelizing compiler with vital performance information of parallel programs. This information may, on the one hand, provide the programmer with a rational basis for deciding on a transformation strategy during an interactive parallelization session; on the other hand, it can be employed to largely automate the transformation process by guiding automatic data distribution generation and the intelligent selection of transformation strategies, based on properties of the parallel program and characteristics of the target architecture.

The underlying approach of P^3T is organized around two major components: *profiling* and *performance parameters*. The original input program is profiled once to derive concrete values for program unknowns such as loop iteration and statement execution counts. Large portions of the profile data can be successfully adapted ([9, 10]) for most important program transformations without redoing the profile run. A parameter based approach is presented to the problem of automatic performance prediction for parallel programs. P^3T models three of the most critical performance aspects of a parallel program: load balance, communication overhead and data locality. A set of parallel program parameters are computed which includes: work distribution, number of transfers, amount of data transferred, network contention, transfer times and number of cache misses. In contrast to many other performance estimators which are restricted to estimated runtimes only, the parallel program parameters as computed by the P^3T can answer two fundamental performance questions:

1. *What performance aspects need to be improved ?*
2. *Where are the program portions requiring performance improvement ?*

The first question is answered, because various parameters reflect different performance aspects. This is in constrast to those approaches which hide all performance parameters in a single estimated runtime figure. P^3T is able to answer the second question, because the parallel program parameters can be selectively determined for statements, loops, procedures, and the entire program; furthermore their effect with respect to individual processors can be examined. P^3T provides vital information about the performance impact of various data distributions and a variety of program transformations. This includes loop interchange, fusion, distribution, scalar forward substitution and inter-loop communication fusion.

The parallel program parameters are designed as machine independent as possible. However in order to build a highly efficient performance estimator we added some of the most important machine specific factors including cache line

size, overall number of cache lines, data type sizes, routing policy, startup times and message transfer time per byte. Much of this information can be easily adapted for a variety of different architectures. The complexity to compute the parallel program parameters is independent of problem size, loop iteration and statement execution counts. As a consequence, the described method is faster than simulating or actually compiling and executing the parallel program.

A graphical user interface is provided which allows filtering of performance data and relating it back to the original parallel code. Color-coded performance visualization enables the user to immediately identify hot spots in the parallel program. The program's performance data can be filtered and displayed at various levels of detail.

P^3T has been developed in the context of the *Vienna Fortran Compilation System (VFCS)*, which is a state-of-the-art parallelization tool for distributed memory systems, and is being successfully used to support the manual and automatic restructuring of programs under this system.

The rest of this paper is organized as follows: Section 2 outlines the underlying compilation and programming model. The next section explains P^3T as an integrated tool of the *VFCS*, and describes the parallel program parameters. Section 4 reports on experimental results. Finally, some concluding remarks are made and future work is outlined.

2 Compilation and Programming Model

P^3T is an integrated tool of the *VFCS* ([6]), which is an automatic parallelization system for massively parallel distributed-memory systems (DMS) translating Fortran programs into explicitly parallel message passing programs. This section describes the underlying compilation and programming model of the *VFCS*.

The parallelization strategy of the *VFCS* is based on domain decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model. This model implies that each processor executes the same program based on a different data domain. The input to *VFCS* are Vienna Fortran programs ([35]). Vienna Fortran is a machine-independent language extension to Fortran77, which provides annotations for the specification of data distributions. Note that for the single sequential profile run to obtain concrete values for program unknowns, the Vienna Fortran language extensions are ignored. Thus the profile run is performed based on the original sequential program. The output of *VFCS* is a parallel Fortran program with explicit message passing. The performance prediction methods described in this paper are implemented for Fortran programs. However, most of the incorporated techniques can be employed for other languages such as the C programming language as well.

Let P denote the set of processors available for the execution of the parallel program, and A an array with *index domain* I^A, as determined by the declaration of A. A *data distribution* for A with respect to P is a total function $\delta^A : I^A \rightarrow \mathcal{P}(P)$, which associates one or more processors with every element of A. If $p \in \delta^A(i)$ for some $i \in I^A$, then a processor p *owns* $A(i)$, or $A(i)$ is local to p. The set of all local variables of a processor p is denoted by $\lambda(p)$; the corresponding projection containing only elements of A is specified by $\lambda^A(p)$. In this paper, only arbitrary *block* distributions, and total replication of arrays are considered. For each p and each A, $\lambda^A(p)$ is called the *segment* of A associated with p. For an m-dimensional array A, $\lambda^A(p)$ can then be characterized by a Cartesian product $\mathrm{X}_{i=1}^{m} D_i^p$, where each $D_i^p = [l_i^p : u_i^p]$ describes the range of index values in dimension i of A that is associated with processor p.

The *work distribution* of a parallel program is determined – based on the data distribution – according to the *owner computes paradigm*. There are no other work distribution concepts considered such as those associated with FORALL loops in Vienna Fortran. The *VFCS* uses masked assignment statements to implement this rule. Let $S : v = \ldots$ denote an assignment statement, where v is a subscripted variable of the form $A(exp_1, \ldots exp_m)$, and A is a properly distributed (non-replicated) array. Then the associated *masked assignment statement* is written as $OWNED(v) \rightarrow S$. The execution of this statement in a process p results in the execution of S iff p owns v. $OWNED(v)$ is called the *mask* of S. For all other statements, the mask is constant TRUE.

Processors can only access local data. Non-local data referenced by a processor are buffered in so-called *overlap areas* that extend the memory allocated for the local segment. The updating of the overlap areas is automatically organized by the system via explicit message passing. This is the result of a series of optimization steps that determine the communication pattern, extract single communication statements from a loop, fuse different communication statements, and remove redundant communication. These optimizations are applied after the first step in the parallelization of

the program has been performed.

For each potentially non-local access to a variable, a communication statement *EXSR* (EXchange Send Receive) is automatically inserted. An *EXSR* statement ([18]) is syntactically described as $EXSR\ A(I_1, ..., I_n)\ [l_1/u_1, ..., l_n/u_n]$, where $v = A(I_1, ..., I_n)$ is the array element inducing communication and $ovp = [l_1/u_1, ..., l_n/u_n]$ is the overlap description for array A. For each i, l_i and u_i respectively specify the left and right extension of dimension i in the local segment. The dynamic behavior of an *EXSR* statement can be described as follows:

```
IF (executing processor p owns v) THEN
      send v to all processors p', such that
      (1) p' reads v, and
      (2) p' does not own v, and
      (3) v is in the overlap area of p'
ELSE IF (v is in the overlap area of p) THEN
      receive v
   ENDIF
ENDIF
```

A single *transfer* refers to the exchange of a data message between two processors.

In order to take procedure calls (Fortran subroutine and function calls) into account, the parameter outcome for a single procedure call instantiation is supposed to be independent of the call site. This means that the parameter outcome at a particular call site is the same as the parameter outcome of the procedure over all call sites, which is a common assumption made for performance estimators ([29]).

In Vienna Fortran ([35]) the user may declare processor arrays by means of the *PROCESSORS* statement. For instance, *PROCESSORS P(4,8)* declares a two-dimensional processor array with 4x8 processors. Distribution annotations may be appended to array declarations in order to specify distributions of arrays to processors. For instance, *REAL A(N,N) DIST(BLOCK,BLOCK) TO P* distributes array A two dimensional block-wise to the processor array P.

3 P^{3}T: A Static Parameter based Performance Prediction Tool

In this section we describe the context in which P^3T is applied in the *VFCS* – see Figure 1. Moreover, the graphically oriented user interface of P^3T is presented. Finally, we outline how to compute the parallel program parameters.

1. *VFCS Frontend*
   In the first step, the Vienna Fortran program is run through the *VFCS* frontend ([6]), which involves extensive intra- and interprocedural analysis and program normalization and standardization. Figure 2 displays the *VFCS* main window after the frontend has been processed.

2. *Weight Finder*
   In the next step, the Weight Finder ([10]), which is an advanced and highly optimized profiler for Fortran programs, is applied. It initiates a single profile run of an instrumented Vienna Fortran program, which obtains concrete values for program unknowns: loop iteration and statement execution counts. A program attributed by these values is created. For the profile run all explicit parallel constructs are ignored (for instance, PROCESSORS and DIST statements) or replaced (for instance, FORALL by DO loop). Note that the profile run is done based on a sequential program. In [9] we have shown that large portions of the profile data can be successfully adapted for most important program transformations without redoing the profile run. However, for changing the problem size, we currently have to repeat the profile run. We are investigating scalability methods such as those proposed in [34] in order to automatically scale profile data based on a small problem size. Program behavior perturbation is not relevant for P^3T, as we solely use the Weight Finder to obtain values for statement execution and loop iteration counts. We believe that profiling is a reasonable choice to handle program unknowns, in particular if we consider the fact that most performance estimators simply apply guessing for program unknowns.

3. *VFCS Parallelization and Optimization Engine*
   Based on the program attributed by the concrete values for program unknowns, the user may select a data

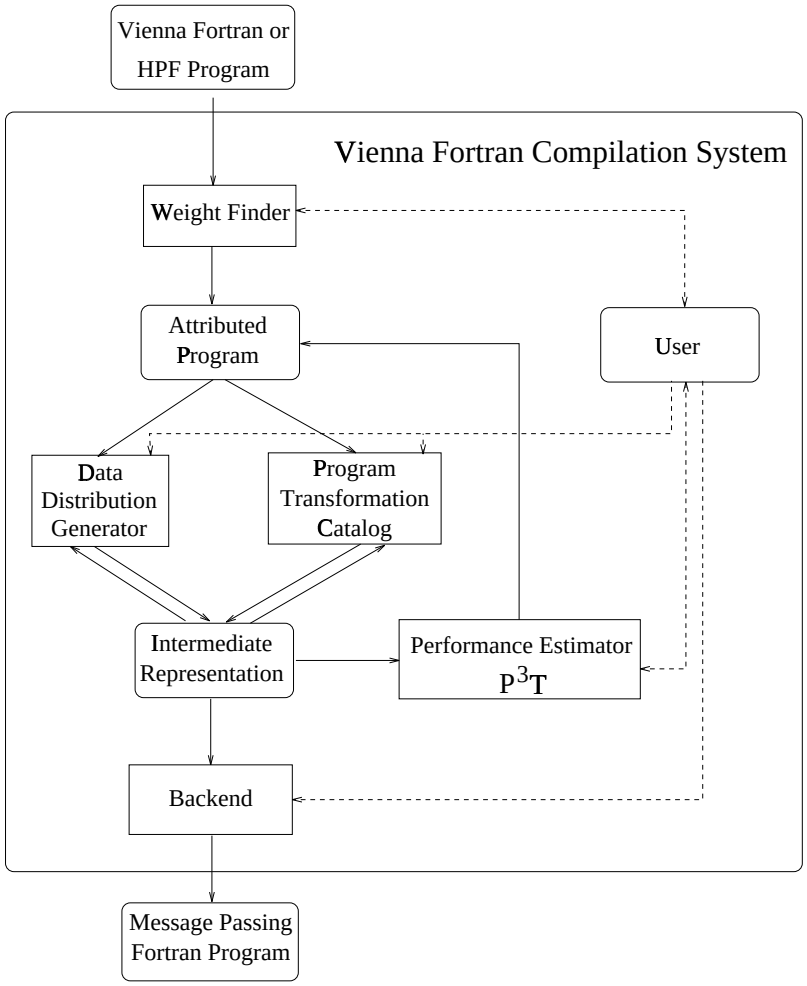


Figure 1: The Structure of P^3T as part of the VFCS

distribution and/or a sequence of program transformations ([9, 12]) to parallelize and optimize the program. A data distribution imposes the parallelism on a program, while program transformations such as loop interchange, distribution, tiling and unrolling are used to tune a program regarding cache performance, communication overhead and load balance. Figure 3 shows the *VFCS Interactive Data Distribution* window, which enables the user to specify and modify the data layout of arrays. The corresponding arrays – F, U, UHELP, and RES – as displayed in this window are distributed two-dimensional block-wise to a 4x4 processor array P. The user can apply program transformations by using the *VFCS Transformations* menu (cf. Figure 4).

4. P^3T

Selecting the P^3T entry in the performance pulldown menu of the *VFCS* main window (Figure 2) presents the P^3T *Options* window (Figure 5). This window enables the user to select the parallel program parameters for various code segments (for instance, DO loops, communication statements, etc.) of interest. The resulting parameter values can be scaled by multiplying them with a factor $v \in \{10^{-9}, \ldots, 10^0, \ldots, 10^9\}$. This can be useful for large parameter values such as number of cache misses, or if the amount of data transferred should be displayed as Mbytes instead of bytes. Selecting the single instantiation parameter type results in the display of the parameters based on a single execution of the selected code segments, while the accumulated type yields accumulated figures regarding the entire program run.

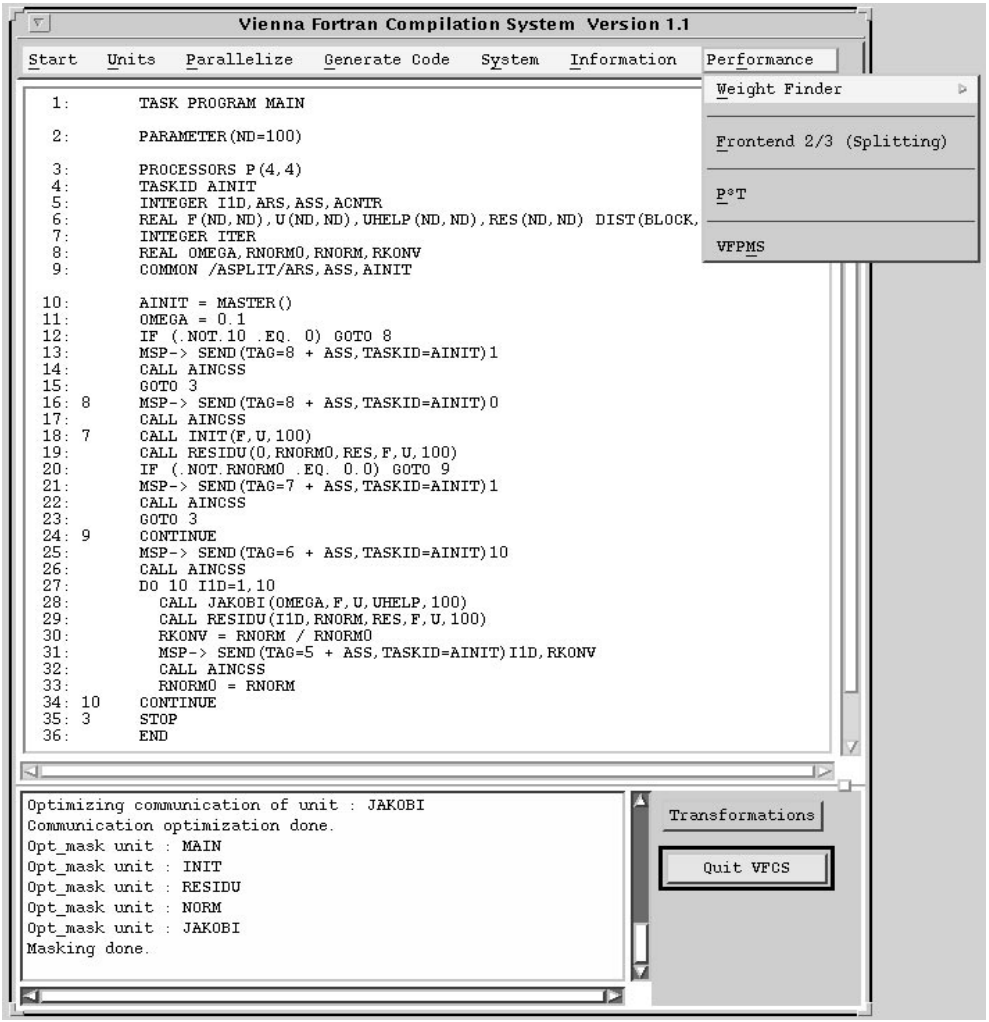


Figure 2: *VFCS* main window after frontend processing

Selecting the *Start* button in the P^3T Options window initiates the computation of the parallel program parameters by P^3T. Based on the parallel program parameters the user is able to identify the location and the nature of performance losses. Figure 6 shows the *VFCS* main window with color-coded performance bars for all selected source code lines. A performance bar is defined by a sequence of sub-bars, each of which corresponds to a specific parameter outcome. TTA, NTA, TDA, NCA, WD and CMA respectively specify the accumulated parameter outcome for transfer times, number of transfer, amount of data transferred, network contention, work distribution, and number of cache misses. The performance goodness is indicated by a range of colors; the brighter the color the better the performance outcome. Dark colors reflect performance losses (hot spots). In addition each sub-bar contains a percentage value which is compared to the worst case performance outcome of the entire program. For instance, the *NTA* value (7.63 %) of source code line 19 indicates that this line accounts for 7.63 % of the accumulated number of transfers with respect to the entire program. The color-coded bar of the first source code line (*TASK PROGRAM MAIN*) corresponds to the performance of the entire program.

Clicking on an arbitrary source code line – for instance, line 28 in Figure 6 – will present a *Performance Data* window (see window at lower right section of Figure 6), which visualizes detailed performance information for that line. For each parameter the precise absolute outcome is visualized. The worst case performance outcome is established by the color-coded bar at the bottom of this window. All parameter bars are compared to this bar. In addition to the color also the length of the bar reflects the performance goodness.

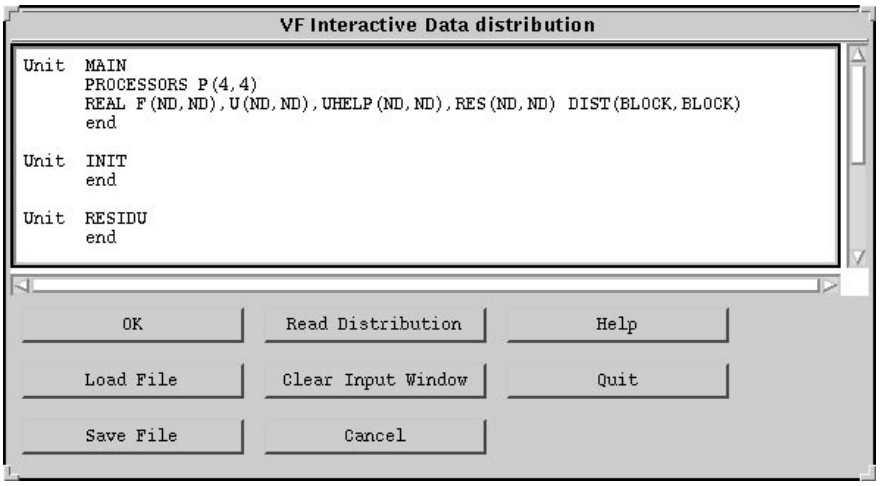


Figure 3: Modification of the data distribution under the *VFCS*

A global performance overview is presented by selecting the *All Units* button in the P^3T *Options* window. A window (Figure 7) pops up, which displays a sorted list of all program units with respect to performance, where the main program, all functions and subroutines define a unique program unit. The sorting is based on a list of parameter priorities, which can be specified by the user. For this example, the transfer time parameter has highest priority. As a consequence, the MAIN program reflects the worst and the INIT subroutine the best performance outcome.

$P^3T's$ graphical user interface permits the user to immediately identify the hot spots of a program. Displaying the parallel program parameters transparently, as opposed to hiding them in a single estimated runtime figure, allows drastic narrowing of the transformation sequence and data partition search space. Moreover, P^3T proposes a set of selected program changes in order to tune the performance with respect to every individual parallel program parameter. For instance, changing the data distribution may improve the program's load balance; loop distribution, privatizing variables, and scalar expansion are useful to decrease the communication overhead; data locality can be gained by strip mining, interchanging, and unrolling of loops, and by modifying the data distribution. The user controls the subsequent application of program changes by the *VFCS* and the estimation of the resulting performance outcome by P^3T, which implies a program parallelization and optimization cycle (cf. Figure 1).

5. *Backend*
   Once P^3T reports reasonable performance gains or the code development time limit is reached, the user selects a specific backend for the desired target architecture, based on which the *VFCS* generates an explicitly parallel and optimized message passing program.

3.1 Parallel Program Parameters

The *parallel program parameters* characterize certain important performance aspects of parallel programs. The parameters as statically computed by P^3T include: work distribution, amount of data transferred, number of data transfers, transfer times, network contention, and number of cache misses. An important objective in the design of these parameters was to extract as much machine independent information out of a parallel program as possible. The parameters work distribution, number of data transfers and amount of data transferred are highly portable across a variety of different target architectures. On the other hand, for those parameters which are partially machine dependent – transfer times, network contention and number of cache misses – we defined a clear interface between dependent and independent machine information. For instance, the only machine knowledge required for the cache misses parameter is the cache line size and the overall number of cache lines available on a specific target architecture.

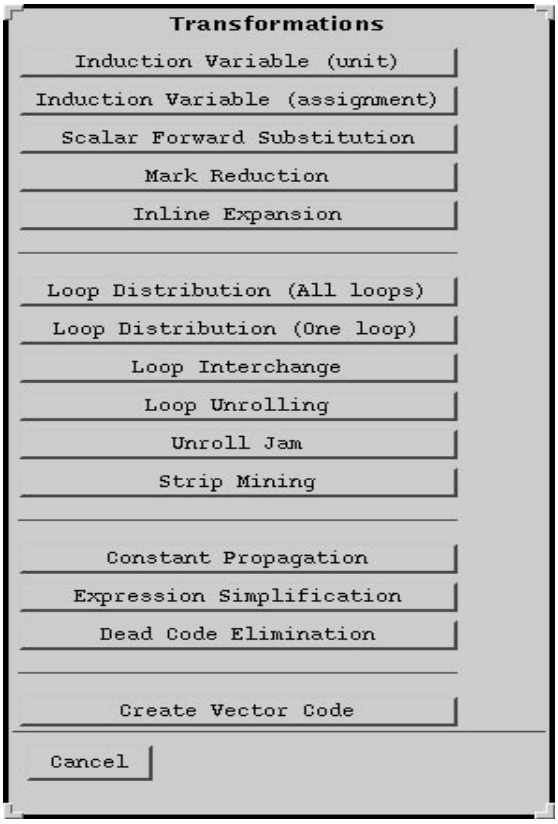


Figure 4: The *VFCS* program transformation catalog

The parallel program parameters are fully implemented for statements, loops, procedures and the entire program; furthermore their effect with respect to individual processors can be examined. All parameters are defined such that a parameter value equal to zero reflects optimal performance outcome, while increasing values indicate increasing performance losses. This facilitates the parameter evaluation significantly. The estimation accuracy of the work distribution, number of transfers, amount of data transferred and transfer time parameters is extremely good. For these parameters the accuracy consistently improves for increasing problem sizes. The number of cache misses and the network contention parameters specify upper bounds. All parameters are briefly outlined in the following. A detailed description, which also shows how to extend the parameters to loops, procedures and an entire program, and an evaluation in terms of accuracy can be found in [9, 8, 11, 13].

3.1.1 Work Distribution (WD)

The *work distribution* defines how even the work implied by a parallel program is distributed across the processors executing the program. This parameter primarily depends on a program's loop iteration spaces and array distributions. Consider the following n-dimensional loop nest L with a masked statement S referencing a m-dimensional array A.

```
DO 10 I1 = B1, E1
   DO 10 I2 = b2(I1), e2(I1)
      DO 10 I3 = b3(I1, I2), e3(I1, I2)
         ...
         DO 10 In = bn(I1, ..., In-1), en(I1, ..., In-1)
            ...
S:          OWNED(A(f1(I1, ..., In), ..., fm(I1, ..., In))->  A(f1(I1, ..., In), ..., fm(I1, ..., In)) = ...
            ...
10 CONTINUE
```

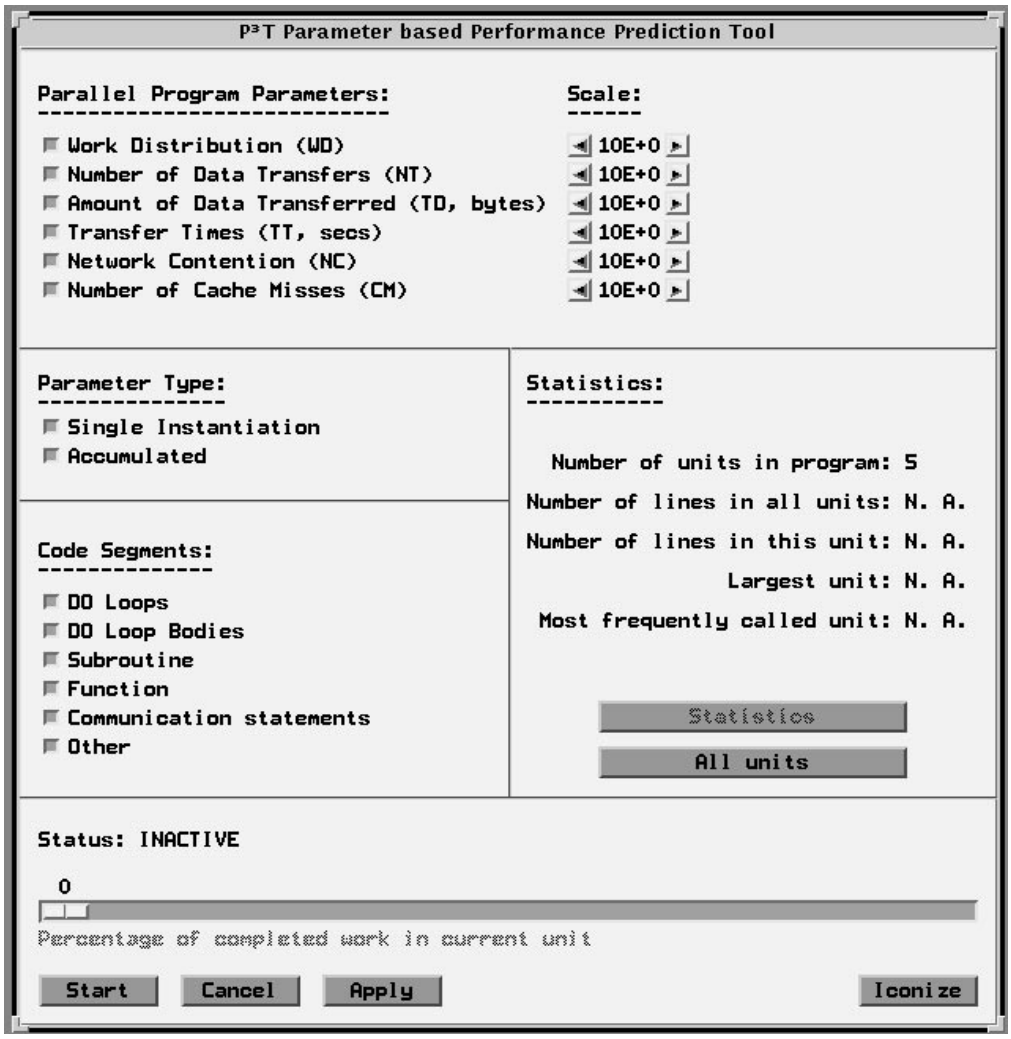


Figure 5: Selecting P^3T Options

$work(S, p)$, the work load for S with respect to processor p, is equal to the number of times the mask of S evaluates to $TRUE$ for p. In order to compute this parameter we observe that L defines a *loop iteration space* which is a n-dimensional polytope Y_s in $\mathbb{R}^n$ and can be described by a set of $2 * n$ inequalities:

$$\begin{array}{rcccl} B_1 & \leq & I_1 & \leq & E_1 \\ b_2(I_1) & \leq & I_2 & \leq & e_2(I_1) \\ & & \cdots & & \\ b_n(I_1, ..., I_{n-1}) & \leq & I_n & \leq & e_n(I_1, ..., I_{n-1}) \end{array}$$

where B_1, E_1 are variables or constants and specify the lower or upper bound, respectively, of I_1. b_i and e_i are functions which respectively specify the lower and upper bound of I_i $(1 \leq i \leq n)$. They linearly depend on $I_1, \ldots, I_{i-1}$.

Figure 6: Source code with P^3T performance data

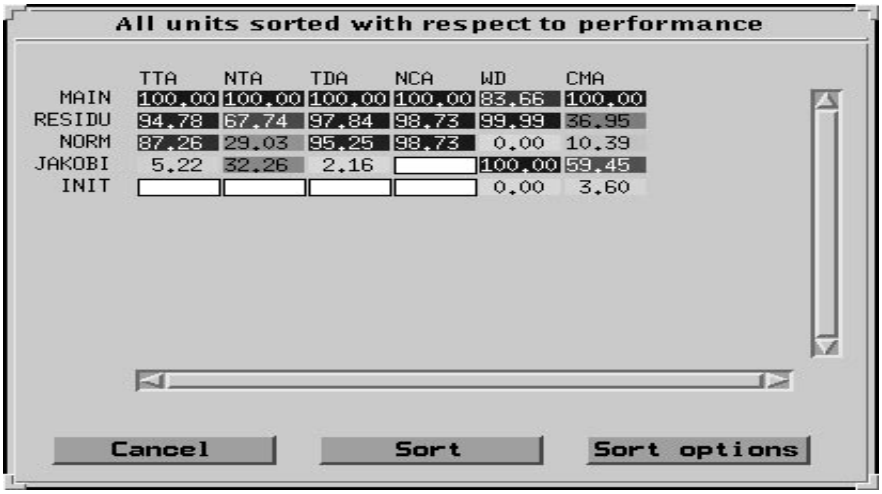


Figure 7: Sorted list of program units with respect to performance

In order to find out how many times a mask evaluates to $TRUE$ for p, the *subscript functions*, which are associated with $A(f_1(I_1,\ldots,I_n),\ldots,f_m(I_1,\ldots,I_n))$ and based on the array segment boundaries of p, are mapped into $\mathbb{R}^n$. This mapping process can be described by a set of $2*m$ inequalities:

$$\begin{array}{rcccl} l_1^p & \leq & f_1(I_1,...,I_n) & \leq & u_1^p \\ & & \cdots & & \\ l_m^p & \leq & f_m(I_1,...,I_n) & \leq & u_m^p \end{array}$$

where f_i $(1 \leq i \leq m)$ is a linear function of I_1^{i-1} (the set of loop iteration variables of the loops from level 1 to $i-1$), and l_i^p (u_i^p) the lower (upper) segment boundary for dimension i of array A associated with p.

Each of the above inequalities defines two n-dimensional half-spaces which may result in an intersection with Y_s. After $2*m$ intersections, a n-dimensional polytope Y_p is created, whose volume serves as an approximation for $work(S,p)$. To be exact, every integer-valued vector in Y_p represents a single evaluation to $TRUE$ for the mask of S with respect to p. The overall number of these integer-valued vectors is the precise amount of work to be done by processor p.

Mapping the subscript functions – based on the boundaries of array A – into $\mathbb{R}^n$ and the resulting intersection with Y_s yields a polytope whose volume serves as an approximation for $work(S,P^A)$, the overall amount of work contained in a statement S.

The intersection and volume algorithm is explained in detail in [9]. For example, Figure 8 illustrates Y_s in $\mathbb{R}^n$ (solid line bounded box), the intersecting half-spaces (dashed lines) and the resulting Y_p (striped area) for a two-dimensional loop nest, with $I_1 = I$ and $I_2 = J$.

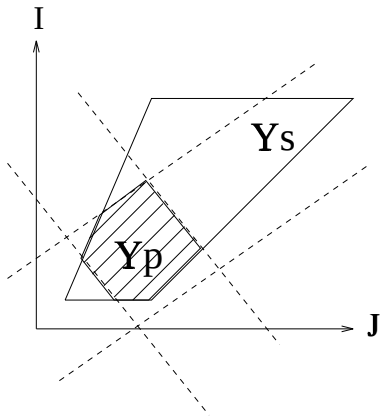


Figure 8: Intersection between the subscript functions and the loop iteration space

The optimal amount of work to be done by every processor in P^A is given by the arithmetic mean:

$$owork(S) = work(S,P^A)/|P^A| \tag{1}$$

with P^A the set of processors to which array A is distributed.

$wd(S)$ – the goodness of the work distribution for S – can now be specified by the standard deviation (σ) divided by the arithmetic mean, which is known as the variation coefficient in statistics:

$$wd(S) = \frac{1}{owork(S)} \sqrt[2]{\frac{1}{|P^A|} \sum_{p \in P^A} \big(work(S,p) - owork(S)\big)^2} \tag{2}$$

with $0 \leq wd(S) \leq |P^A| - 1$. The work distribution goodness deteriorates for increasing values of $wd(S)$. If the work is evenly distributed across all processors in P^A, then $wd(S) = 0$; in case of an array replication $wd(S) = |P^A| - 1$.

The goodness of the work distribution for a loop L containing a sequence of statements can now be specified as follows:

$$wd(L) = \sum_{S \in \varrho(L)} \frac{freq(S) * wd(S)}{\sum_{S \in \varrho(L)} freq(S)} \tag{3}$$

Here, $\varrho(L)$ is the set of assignment statements in L.

3.1.2 Amount of Data Transferred (TD)

Let P^A denote the set of processors involved in a specific communication statement C implied by a distributed array A. Then we define $td(C)$ as the number of data bytes transferred among all processors in P^A. Consider the following loop kernel L:

```
        DO I=B1,E1
          EXSR A(I,*) [1/0,1/0]
          DO J=b2(I),e2(I)
S:           OWNED (A(I,J)) → A(I,J) = A(I,J) - A(I-1,J-1)
          ENDDO
        ENDDO
```

In general, L defines an iteration space, which is a n-dimensional polytope Y_s in $I\!R^n$. The execution of S in processor p accesses non-local elements. These must be communicated to p; their values are stored in the overlap area associated with $\lambda^A(p)$. In Figure 9.a, processor p receives the boundary elements of its associated overlap area – shaded area – from processors p', p'', and p'''.

$td^p(C)$, the overall amount of data transferred for a communication statement C with respect to p, is computed as follows:

1. The array access subscript functions – based on the array segment boundaries extended by the overlap areas for processor p – are mapped into $I\!R^n$ and build the intersection with Y_s. This yields an intersection body Y_1 (union of solid striped and dashed striped area in Figure 9.b).

2. The subscript functions – based on the array segment boundaries of processor p – are mapped into $I\!R^n$. The resulting intersection with Y_s yields Y_2 (dashed striped area in Figure 9.b).

3. The difference between Y_1 and Y_2 defines a polytope $Y_p = Y_1 - Y_2$ (solid striped area in Figure 9.b).

Every n-dimensional integer-valued vector in Y_p represents a single data element to be received by p from its neighboring processors. The precise amount of data transferred is therefore the number of all such vectors inside of Y_p. The approximate amount of data transferred is the volume of Y_p, which we denote by $Vol(Y_p)$; furthermore $td^p(C) = Vol(Y_2) - Vol(Y_1)$.

If the overlap areas of p extend the corresponding local array segment in only one array dimension, then the effort to compute $td^p(C)$ can be substantially decreased by mapping the subscript functions – based on the overlap area of p with respect to A – into $I\!R^n$. The volume of the resulting intersection body yields exactly the same $td^p(C)$ as the previous algorithm but with half the computational effort. Finally, the approximate value for the number of data elements received by all processors involved in a specific communication statement C is defined by:

$$td(C) = \sum_{p \in P^A} td^p(C) \tag{4}$$

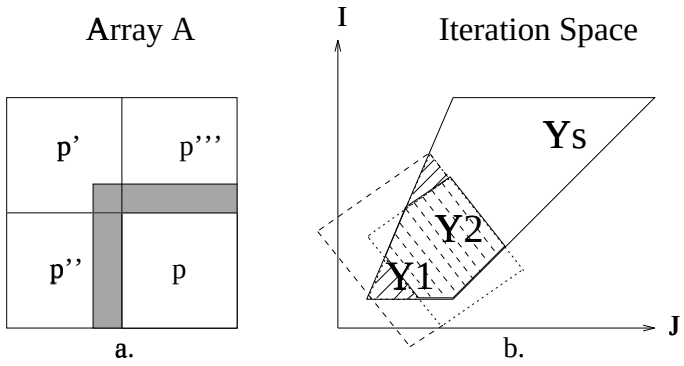


Figure 9: a. 2-dimensional block distribution of array A with overlap area of processor p; b. Intersection between the subscript functions and the iteration space

3.1.3 Number of Data Transfers (NT)

A critical parameter for current parallel architectures is the number of data transfers induced by all processors involved in a communication statement C. We denote this parameter by $nt(C)$. Previous research judges communication statements at a very coarse grain level. In [19] information is provided regarding whether a specific communication statement represents a single transfer, broadcast, many-to-many transfer, etc. However, many approaches do not consider the importance of the loop iteration space. Consider the following loop kernel:

```
                PROCESSORS P(2,2)
                INTEGER A(N,N) DIST(BLOCK,BLOCK)
                ...
                DO I=1,N
                   DO J=1,N/2-1
C:                    EXSR A(I,J-1) [0/0,1/0]
                      OWNED (A(I,J))→A(I,J) = A(I,J-1) + 1
                   ENDDO
                ENDDO
```

Without considering the loop iteration space, one would assume that the communication statement C induces heavy communication, as it is inside of the innermost loop. However, by modeling the loop iteration space, it can be concluded that the loop does not iterate across the segment boundaries of the second array dimension. Consequently there is no communication at all.

$nt(C)$ fully incorporates the loop iteration space and therefore yields very accurate communication information. There are three different communication statement categories for a loop nest L: First, communication statements outside of L (outside communication), which represent the loosely synchronous communication model ([16]), secondly, communication statements inside of the innermost loop of L (inside communication), and third, communication statements between the innermost and outermost loop of L (interior communication). The last two categories represent the asynchronous communication model.

3.1.3.1 Outside Communication An outside communication statement C induces communication on the part of all involved processors in parallel. Let $p \in P^A$ denote the processor which receives the maximum number of messages from other processors. Processor p dominates the communication overhead regarding C. The overlap areas of p can be partitioned into segments associated with different neighboring processors of p ([18, 9]). For every such segment, communication with the associated processor is induced. The approximate value for the number of transfers implied by C is defined as follows:

$$nt(C) = MAX_{p' \in P^A} N_{seg}(p') \tag{5}$$

$N_{seg}(p')$ is the number of different segments for p'.

3.1.3.2 Innermost Communication Communication in the innermost loop is most expensive due to data dependencies which prevent hoisting communication statements out of a loop nest. Every single data element in the overlap area of a processor p with respect to a communication statement C results in a unique transfer. For this case we proceed exactly as in Section 3.1.2. Mapping the subscript functions associated with the referenced array into $\mathbb{R}^n$ and the following intersection with Y_s, the iteration space of L, yields a n-dimensional polytope Y_p. The volume of Y_p serves as the approximation for $nt^p(C)$, the number of transfers for p with respect to C. The approximate value for the number of transfers for all processors induced by C is defined as follows:

$$nt(C) = \sum_{p \in P^A} nt^p(C) \tag{6}$$

3.1.3.3 Interior Communication Consider the following loop kernel:

```
DO I1=2,N1
    DO I2=2,N2
        EXSR A(I1,I2-1,*) [0/0,1/0,1/0]
        DO I3=2,N3
            OWNED(A(I1,I2,I3))→A(I1,I2,I3) = A(I1,I2,I3) - A(I1,I2-1,I3-1)
        ENDDO
    ENDDO
ENDDO
```

$nt^p(C)$, the number of transfers for communication statement C with respect to processor p for the innermost two loops, is derived as follows: We map the array subscript functions associated with the referenced array – based on the array segment extended by the overlap areas for p – into $\mathbb{R}^2$. This results in an intersection with the iteration space of L_2^3, which yields a 2-dimensional intersection polytope Y_p (solid line bounded box in Figure 10.a). The loop L_2 immediately enclosing C specifies the communication direction, which is uniquely defined by the $I2$ axis (see Figure 10.a). For the sake of simplicity, if Y_p is not empty, then for every iteration of L_2, processor p receives a message from its neighboring processor. Therefore $\overline{nt^p}(C)$, which is the longest distance (see solid line arrow in Figure 10.a) between any two vectors in Y_p along communication axis $I2$, is an approximate value for the number of transfers for p with respect to C considering L_2^3 only.

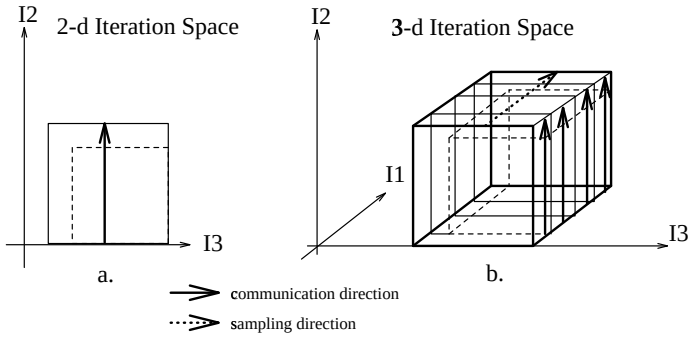


Figure 10: a. Intersection with a 2-dimensional iteration space; b. Intersection with a 3-dimensional iteration space

All loops in L enclosing the communication loop L_2 are sampling loops. In theory, for every iteration of every sampling loop the algorithm as mentioned above has to be performed to compute all $\overline{nt^p}(C)$ values. For this purpose, sampling along the induced axis of the sampling loops needs to be done (see dotted arrow in Figure 10.b). Summing up all $\overline{nt^p}(C)$ for every sample, yields $nt^p(C)$. The sum of all $nt^p(C)$ for all processors involved in C yields $nt(C)$, the overall estimated number of transfers for C.

A variety of computational shortcuts ([9]) are provided to decrease the computational effort of the sampling mechanism. First, for every sampling direction induced by a specific sampling loop L_s, there exists a maximum distance between

any two vectors of Y_p along this direction, which is defined by Dist(L_s). Multiplying $\overline{nt^p}(C)$ by the product of all Dist(L_s) can then be employed as an upper bound for $nt^p(C)$. Secondly, a different step size than one for sampling along L_s is incorporated. The actual step-size chosen depends on Dist(L_s). Both methods however induce a slight reduction in accuracy.

3.1.4 Transfer Times (TT)

This section describes $tt(C)$, which reflects the transfer time implied by a communication statement C. Various performance aspects are incorporated in the model for this parameter: number of transfers, amount of data transferred, several machine specific parameters such as message startup overhead, message transfer time per byte, byte sizes for different data types, and processor distances. The transfer time parameter does not account for network contention. Nevertheless, this performance aspect is computed as a separate parameter in Section 3.1.5.

If C is a communication statement which implies the transfer of data messages due to a reference to a distributed array A in a parallel program Q, and $OA(A,p)$ is the overlap area of a processor $p \in P^A$ with respect to C, then the set of *neighboring segments* of p with respect to C is defined as follows:

$$NS(C,p) := \{K | K = OA(A,p) \cap \lambda^A(q) \text{ for some } q \in P^A \text{ and } p \neq q; K \neq \phi\}$$

Processor p can only receive data from one of its neighboring segments with respect to C. Neighboring segments are inherently owned by some neighboring processor $q \in P^A$ $(p \neq q)$.

For the underlying machine model – a hypercube network topology – the communication time in μsecs for a single transfer between two processors is defined by: $\alpha + \beta * ms + \lambda * hops$, where α is the message startup time, β the message transfer time per byte, ms the size of the transferred data in bytes, and λ the overhead time for every network hop.

In the following we describe how to derive $\overline{tt}(C)$, an estimated value for $tt(C)$, where C is an outside communication statement and A is the distributed array referenced by C:

1. Find processor $p \in P^A$ with maximum $|NS(C,p)|$. This processor is referred to as p_{max}.
2. Classify the neighboring segments in $NS(C,p_{max})$ with respect to the number of network hops. This yields $NS(C,p_{max}) = \{NH_1, \ldots, NH_k\}$, where NH_i $(1 \leq i \leq k)$ is the set of neighboring segments in $NS(C,p_{max})$, which require i network hops to be transferred to p_{max}.
3. Depending on the size of $td^{p_{max}}(C)/nt^{p_{max}}(C)$, we choose the correct set of values for α, β and γ – they may vary for different volumes of data transferred – to compute an estimated value for $tt(C)$:

$$\overline{tt}(C) := \sum_{NH_i \in NS(C,p_{max})} (\alpha + \beta * \frac{td^{p_{max}}(C)}{nt^{p_{max}}(C)} + \gamma * i) * \frac{|NH_i|}{nbs} * nt^{p_{max}}(C) \quad (7)$$

where $td^{p_{max}}(C)$ is the amount of data transferred (cf. Section 3.1.2) and $nt^{p_{max}}(C)$ the number of transfers induced by p_{max} (cf. Section 3.1.3) with respect to C, and $nbs := \sum_{NH_i \in NS(C,p_{max})} |NH_i|$

The above formula for $\overline{tt}(C)$ assumes that the amount of data transferred and the number of transfers for p_{max} are evenly distributed across all neighboring segments in $NS(C,p_{max})$. To simplify this assumption we would have to compute the amount of data transferred and the number of transfers based on neighboring segments at the cost of additional computational complexity. Computing $\overline{tt}(C)$ for interior or inside communication statements is very similar to the above algorithm. For details refer to [9].

3.1.5 Network Contention (NC)

It has been shown ([5, 4, 28, 26]) that network contention, which refers to the sharing of a communication channel by two or more transfers, may severely degrade all network traffic. For example, Shahid Bokhari reports in [4, 3] that on the iPSC/860 hypercube, which uses circuit-switched communications with e-cube routing, channel contention can increase the time to communicate by a factor of more than seven. However, it is very difficult for a programmer to analyze a program regarding network contention. This requires detailed knowledge of the underlying architecture and

parallelizing compiler.

In order to compute $nc(C)$, the network contention implied by a communication statement C, the following assumptions are made:

- A fixed and static routing policy such as the e-cube routing ([9]) on a hypercube topology, which permits specifying of the exact network path of a message, is assumed.
- Network contention occurs, iff two or more transfers occupy at least one specific network channel at the same time in the same direction (*channel contention*). A channel refers to a physical link between two processors on a network topology.

The following describes how to derive $\overline{nc}(C)$, an estimated value for $nc(C)$, where A is the distributed array referenced by C; $ct^+(c)$ and $ct^-(c)$ are the number of transfers respectively traversing a channel c in the positive and negative channel direction. The transfers induced for every $p \in P^A$ with respect to C are supposed to be evenly distributed across all neighboring segments of p, namely $NS(C,p)$. For a precise analysis the exact number of transfers and associated amount of data transferred would have to be determined for each neighboring segment at the cost of extra computational complexity ([9]). $nt^p(C)$ denotes the number of transfers implied by a processor p with respect to C. This figure is equal to the number of receive operations processed by p. [9, 13] explains in detail how to statically estimate $nt^p(C)$ with high accuracy.

1. Initialize NC, a network contention counter, and $ct^+(c)$, $ct^-(c)$ for all channels c with zero
2. For every $p \in P^A$ do the following:
   (a) Compute $NS(C,p)$, the set of all actually accessed neighboring segments of p, $nt^p(C)$, the number of transfers for p induced by C, and $w = nt^p(C)/NS(C,p)$.
   (b) For each $h \in NS(C,p)$ – which refers to a data portion to be received by p – find the owning processor p' (responsible to send h to p), and derive the exact message transfer path between p' and p according to the e-cube routing scheme ([9]); for each network channel c traversed, the traversal direction is evaluated; For a positive channel traversal direction: if $ct^+(c) = 0$, which means that the channel is assumed to be free, then $ct^+(c) := w$ (w messages are supposed to traverse c consecutively in the positive direction) without changing NC; otherwise c is assumed to be already occupied due to previous transfers. Therefore, NC is incremented by MIN$(w, ct^+(c))$ number of channel conflicts and $ct^+(c)$ is set to $|ct^+(c) - w|$ in this order; For a negative channel traversal direction this step is done for $ct^-(c)$ respectively;
3. NC is multiplied by a probability value φ ($0 \leq \varphi \leq 1$).

φ in the above algorithm specifies a probability value between 0 and 1, which depends on C. For communication statements outside of a loop nest the loosely synchronous communication model is assumed, which means that all processors are supposed to interact simultaneously. For those inside of a loop nest, which represent the asynchronous communication case, there is a lower probability that all processors communicate in concert. Therefore, if C is an outside communication statement, then φ is supposed to be rather large (for instance, 0.8), otherwise it is assumed to have a small value (for instance, 0.2); The actual value for φ strongly depends on the application. It might well be the case that the processors executing a parallel program do not process a loop L at the same time during execution. In such a case φ should be obviously smaller than 0.8 with respect to communication statements outside of L. $\overline{nc}(C)$ is defined by the value of NC at the end of the above algorithm.

Note that the estimated number of channel contentions is an upper bound. This is because time constraints are ignored. If two transfers ever occupy the same channel in the same direction based on the static e-cube routing then they are supposed to induce a single channel contention. This ignores the fact that channel contention only occurs, if both transfers traverse the same channel at the same time. Only precise and therefore expensive simulation techniques or actually executing – supposing that a channel contention measurement tool is available – the parallel program would allow deducing of the exact channel contention behavior.

3.1.6 Number of Cache Misses (CM)

It is well known ([8, 32, 25, 23, 14]) that inefficient data mapping into the memory hierarchy (data locality problem) may cause major program performance degradation. This section describes how to estimate the number of accessed cache lines, which correlates with the number of cache misses.

The principal idea is to group all array references into array access classes ([9]) such that all references in a specific class exploit reuse of array elements in the same set of array dimensions. The definition of array access classes is based on a specific number of innermost loops of a not necessarily perfectly nested loop L. Two array references are in the same array access class regarding loop nest L_l^n – the set of $n-l+1$ innermost loops in loop nest L – if they actually access some common memory location in the same array dimensions and reuse occurs in L_l^n. The common accesses occur on either the same or a different iteration of L_l^n.

In order to calculate $\overline{cmL^p}(L)$, the estimated number of cache misses for a specific processor $p \in P$ with respect to L, an iterative algorithm is incorporated. First, this algorithm computes cl_n, the estimated number of cache lines accessed by p in L_n^n, the innermost loop of L. If cl_n is greater than OCL – the overall number of cache lines available in the underlying architecture – then executing all iterations of the innermost loop does not require all cache lines available on the target processors p. The reference groups of the innermost loop are re-grouped into array access classes for L_{n-1}^n, which yields cl_{n-1}, the number of cache lines accessed in the innermost 2 loops of L. The iterative algorithm continues to compute cl_i $(1 \leq q \leq i \leq n)$ for the next higher loop levels until cl_q (L_q is defined as the *cache overflow loop*) is less then OCL. Then the current cl_q is multiplied by the loop iteration count product of all loops in L_1^{q-1}. This results in an upper bound for the overall estimated number of cache lines accessed – which is assumed to correlate with the number cache misses – in L:

$$\overline{cmL^p}(L) = cl_q * \prod_{i=1}^{q-1} iter(L_i) \tag{8}$$

$iter(L_i)$ specifies an average number of loop iterations for L_i, as derived by the Weight Finder ([10]). For a parallel program $iter(L_i)$ has to be adapted which is shown in [9, 8]. Note that the above equation is also applicable to a sequential program ($|P| = 1$).

Assume that there is a set of h_k array access classes for L_k^n. In the following we show how to compute for every ac $\in h_k$ the associated cl_k^{ac}, the number of estimated cache lines accessed by all references in ac with respect to L_k^n:

1. If all references in ac are loop invariant with respect to I_k in all array dimension subscript expressions, then $cl_k^{ac} = cl_{k+1}^{ac}$.
2. If for all array references in ac there exists at least one array containing I_k in the first array dimension subscript expression and there does not exist an array in ac containing I_k in any dimension subscript expression other than the first one, then $cl_k^{ac} = (cl_{k+1}^{ac} * iter(L_k))/cls + 1$. In this equation *cache line overlap* ([14]) is modeled by adding 1 to cl_k^{ac}. cls specifies the cache line size in bytes.
3. otherwise $cl_k^{ac} = cl_{k+1}^{ac} * iter(L_k)$

The references of several array access classes $\{ac_1, \ldots, ac_q\}$ at L_{k+1} can be merged to a new array access class ac_r at L_k, such that $cl_{k+1}^{ac_r} = MAX(cl_{k+1}^{ac_1}, \ldots, cl_{k+1}^{ac_q})$. It is important that the maximum – instead of the sum – of the number of cache lines accessed across all array access classes is computed, as the references merged, are supposed to exploit data reuse.

4 Experiments

The FLO52 program of the Perfect Benchmarks ([7]) analyzes the transonic flow past an airfoil by finding a solution of the unsteady Euler equations. In this section, we evaluate the usefulness of P^3T in the effort of guiding the *VFCS* to optimize and parallelize the EFLUX subroutine, which is one of the most time consuming and interesting subroutines – containing both 2 and 3-dimensional arrays – in the FLO52 program. For our experiments we used the *VFCS* Version 1.1 to generate parallel program versions by applying a variety of program transformations and

data distribution strategies, the *Weight Finder* to derive the sequential program parameters, the P^3T to statically compute the parallel program parameters, and the iPSC/860 hypercube to run the parallel programs for validation of our predicted results. The iPSC time and event measurements were obtained on a 16 node Intel iPSC/860 hypercube. It is running under release 3.3.1 of the Intel software. Each program was compiled under -O4 using Release 3.0 of if77, the iPSC/860 Fortran compiler. Time measurements were made using dclock(), a microsecond timer.

4.1 Selecting a data distribution strategy

We evaluated three different data distribution strategies for a specific problem size:

v1: Distribute the second dimension of all arrays contained in the EFLUX program to a one-dimensional processor array of 16 processors; this corresponds to a column-wise distribution.

v2: Distribute the first dimension of all arrays contained in the EFLUX program to a one-dimensional processor array of 16 processors; this corresponds to a row-wise distribution.

v3: Distribute the first 2 dimensions of all arrays contained in the EFLUX program to a 4x4 processor array; this corresponds to a two-dimensional block-wise distribution.

Figure 11 displays the three program versions with their associated measured runtime MT (indicated in seconds) and the parallel program parameters as computed by P^3T. If we assume that for the iPSC/860 hypercube, it is the transfer time TT (indicated in μsecs), which dominates the program performance, then we obtain the same performance ranking for all three program versions regarding MT and TT. For both MT and TT, v1 reflects the best, v2 the second best and v3 the worst performance. Note that both v2 and v3 have almost identical number of transfers. Despite that, the transfer time TT, which models network distances (number of network hops), number of transfers (NT) and the amount of data transferred (TD), clearly indicates the better communication behavior for v2. This experiment demonstrates also the importance of the fine grain communication information as provided by the transfer time TT. In addition, the work distribution (WD) and network contention (NC) of v2 is much better than that of v3. v1 is the best program version due to its superior TT, NT and TD figures. This experiment demonstrated $P^3T's$ ability to find the best among all applied data distributions.

4.2 Applying scalar forward substitution

In order to reduce redundant computation in the original EFLUX code, scalar variables are incorporated as temporary variables to store common subexpressions. These scalar variables are used as part of the right-hand side of array assignment statements. This technique implies several severe performance drawbacks, as the VFCS requires every processor of a parallel program to execute all assignments to scalar variables:

- The masks associated with array assignment statements are not propagated to the loop bounds (loop strip mining – [6]).
- Even though the array assignment statements are executed according to the *owner computes paradigm*, the corresponding common subexpressions are evaluated at every statement instantiation.
- Worst case assumptions about dependences are made, which may significantly increase the communication overhead. In this example it does. Compare the values for NT before and after scalar forward substitution.

For this experiment the scalar assignment statements inside of loops imply communication across the entire processor array domain. In order to prevent this reduction in performance, the VFCS provides the user with an important program transformation, namely *Scalar Forward Substitution* (cf. Figure 4). This transformation replaces – under appropriate conditions ([36]) – an applied occurrence of a scalar variable by the expression on the right-hand side of an assignment to that variable. We therefore applied scalar forward substitution to all three program versions, which decreases the corresponding runtime by approximately one order of magnitude. v1' – the program version based on column-wise distribution – displays still the best performance. However, now the two-dimensional block-wise distribution performs better than the row-wise distribution, which is attributed to its superior TD and TT parameter outcome.

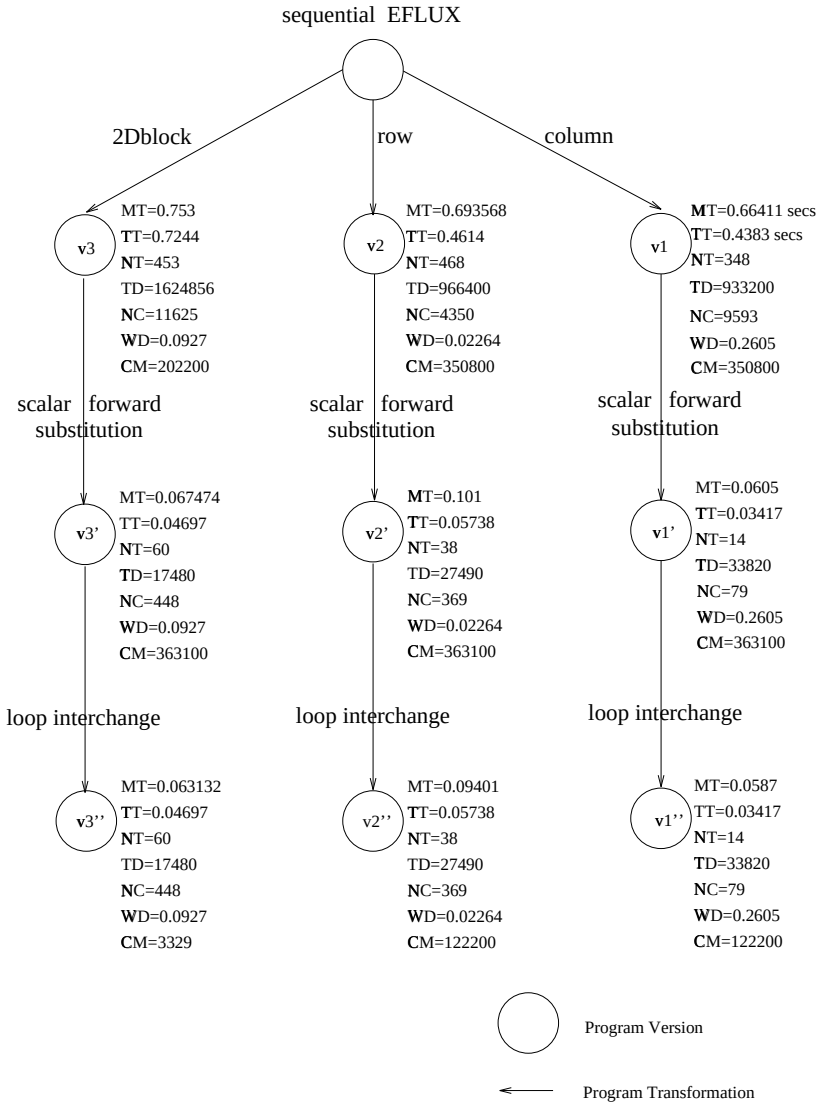


Figure 11: Performance tuning of the EFLUX program

4.3 Applying loop interchange

In order to decrease the high number of cache misses (CM parameter) loop interchange is applied. Thereafter, the cache misses are reduced by approximately 70 %. This effect is also reported by the improved measured runtime MT. If the cache fills up in the innermost loop, then CM of the innermost loop is multiplied with the loop iteration count product of all enclosing loops. As the cache commonly fills up in the innermost loop, the program versions rarely differ at the same horizontal level.

This experiment demonstrated that P^3T successfully

1. supported the *VFCS* in the selection of an efficient data distribution strategy and in the application of two important program transformations: scalar forward substitution and loop interchange.
2. ranked all program versions with respect to their actual performance behavior. This ranking was validated against actually measured runtimes for every program examined.

P^3T is able to help the programmer decide whether a specific program transformation is profitable or not for many other transformations ([12, 9]) such as variable privatizing, loop unrolling, loop peeling, statement mask propagation,

inter-loop communication fusion etc. The programmer may proceed tuning the program's performance by using P^3T until the outcome of all parallel program parameters is sufficiently good – close to zero according to Section 3 – or program development time limits are reached.

5 Conclusion

A usable yet efficient performance estimator for parallel programs is needed to build second generation parallelizing compilers, which require guidance for performance tuning of parallel programs. P^3T is, to the best of the authors knowledge, the first performance estimator to support the user in evaluating the performance outcome of both data distribution strategies and program transformations.

P^3T statically computes a set of parameters that characterize the behavior of parallel programs. This includes work distribution, number of data transfers, amount of data transferred, transfer times, network contention, and number of cache misses. These parameters can be selectively determined for statements, loops, procedures, and the entire program; furthermore, their effect with respect to individual processors can be examined. The computational complexity of the parallel program parameters is independent of the program's problem size, statement execution and loop iteration counts. As a consequence, estimating the parallel program parameters is considerably faster than simulating or actually executing the program.

A graphical user interface is provided which allows filtering of performance data and relating it back to the original parallel code. Color-coded performance visualization enables the user to immediately identify hot spots in the parallel program. The program's performance data can be filtered and displayed at various levels of detail.

The estimator is limited to regular programs. It cannot be employed for irregular problems, which require runtime analysis. Shifting performance estimation into runtime to support runtime optimization will be addressed in future work. P^3T is currently being extended to estimate the performance of High Performance Fortran (HPF - [21]) programs using a separate frontend under the *VFCS*. Ongoing work to fine tune the estimator for a larger set of optimizing transformations and to evaluate it for several other distributed memory architectures will further enhance the usefulness of P^3T. Additional information on our performance prediction work in particular the P^3T, including an electronic copy of this and other papers, can be found at: **http://www.par.univie.ac.at/inst/staff/tf.html**.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *3rd ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Williamsburg, VA, April 21-24 1991.

[2] G. Balbo, S. Donatelli, and G. Franceschinis. Understanding parallel program behavior through petri net models. *Journal of Parallel and Distributed Computing*, 15:181–187, 1992.

[3] S. Bokhari. A network flow model for load balancing in circuit-switched multicomputers. NASA Contractor Report 182049, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, May 1990.

[4] S. Bokhari. Communication overhead on the intel iPSC-860 Hypercube. ICASE Interim Report 10, NASA Contractor Report 182055, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, May 1990.

[5] S. Bokhari. Complete Exchange on the iPSC-860. ICASE Report No. 91-4, NASA Contractor Report 187498, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, January 1991.

[6] B. Chapman, S. Benkner, R. Blasko, P. Brezany, M. Egg, T. Fahringer, H.M. Gerndt, J. Hulman, B. Knaus, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova, and H.P. Zima. *VIENNA FORTRAN Compilation System - Version 1.0 - User's Guide*, January 1993.

[7] The Perfect Club. The perfect club benchmarks: Effective performance evaluation of supercomputers. International Journal of Supercomputing Applications, 1989.

[8] T. Fahringer. Automatic Cache Performance Prediction in a Parallelizing Compiler. In *Proc. of the AICA'93 - International Section*, Lecce, Italy, September 1993.

[9] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, Institute of Software Technology and Parallel Systems, October 1993.

[10] T. Fahringer. The Weight Finder, An Advanced Profiler for Fortran Programs. In *Automatic Parallelization, New Approaches to Code Generation, Data Distribution, and Performance Prediction.* Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, March 1993.

[11] T. Fahringer. Automatically Estimating Network Contention of Parallel Programs. In *Proc. of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Vienna, Austria*, May 1994.

[12] T. Fahringer. Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. In *IEEE Proc. of the Scalable High Performance Computing Conference 1994, Knoxville, TN*, May 1994.

[13] Thomas Fahringer and Hans Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Invited Paper, In Proc. of the 7th ACM International Conference on Supercomputing 1993*, Tokyo, Japan, July 1993.

[14] J. Ferrante, V. Sarkar, and W. Trash. On estimating and enhancing cache effectiveness. In *Proc. of the 4th Workshop on Languages and Compilers for Parallele Computing*, Santa Clara, CA, Aug 1991.

[15] A. Ferscha. A petri net approach for performance oriented parallel program design. *Journal of Parallel and Distributed Computing*, 15:188–206, 1992.

[16] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. and 2. Prentice Hall, Englewood Cliffs, NY, 1988.

[17] A. Gemund. Performance Prediction of Parallel Processing Systems: The Pamela Methodology. In *In Proc. of the 7th ACM International Conference on Supercomputing 1993*, Tokyo, Japan, July 1993.

[18] H.M. Gerndt. *Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[19] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1992.

[20] F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In *Computer Performance Evaluation '92: Modeling Techniques and Tools*, pages 197–210, 1992. (Ed.) R. Pooley and J. Hillston.

[21] High Performance FORTRAN Language Specification. Technical Report, Version 1.0, Rice University, Houston, TX, May 1993.

[22] H. Jonkers. Queueing Models of Parallel Applications: The Glamis Methodology. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools*, pages 123–138, Vienna, Austria, May 1994. Lecture Notes in Computer Science, Springer Verlag.

[23] K. Kennedy and K.S. McKinley. Optimizing for Parallelism and Data Locality. In *International Conference on Supercomputing 1992*, pages 323–334, Washington D.C., July 1992.

[24] K.Y.Wang. A Framework for Static, Precise Performance Prediction for Superscalar-Based Parallel Computers . In H. Sips, editor, *In Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, pages 413–427, Delft, The Netherlands, December 1993.

[25] M. Lam, E. Rothberg, and M. Wolf. The cache performance and Optimizations of Blocked Algorithms. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara,CA, April 1991.

[26] G. Pfister and V.A. Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. In *Proc. of the 1985 International Conference on Parallel Processing*, pages 790–797, 1985.

[27] J.P. Prost and S. Kipnis. A Multilevel Trace-Driven Simulation Approach for Performance Analysis of Distributed-Memory Programs. RC 17612 (#77650), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, January 1992.

[28] D. Russel. *The Principles of Computer Networking.* Cambridge University Press, Cambridge, Great Britain, 1991.

[29] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessor.* The MIT Press, Cambridge, Massachusetts, 1989.

[30] K. Trivedi, R. Haverkort, A. Rindos, and V. Mainkar. Techniques and Tools for Reliability and Performance Evaluation: Problems and Perspectives. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools*, pages 1–24, Vienna, Austria, May 1994. Lecture Notes in Computer Science, Springer Verlag.

[31] K.Y. Wang. A Performance Prediction Model for Parallel Compilers . Technical report, Computer Science Dept., Purdue University, November 1990. Technical Report CSD-TR-1041, CAPO Report CER-90-43.

[32] M.E. Wolfe and M. Lam. A data locality optimizing algorithm. In *In Proceedings of the SIGPLAN 91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[33] N. Yazici-Pekergin and J.M. Vincent. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Transactions on Software Engineering*, 17(10):197 – 217, October 1991.

[34] X. Zhang, Y. Yan, and K. He. Latency metric: an experimental method for measuring and evaluating parallel program and architecture scalability. *Journal of Parallel and Distributed Computing*, 22(3):392–410, 1994.

[35] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. Technical report, ICASE, Hampton,VA, 1992. ICASE Internal Report 21.

[36] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison-Wesley, 1990.

Locality Analysis and Optimization for SMPs

Guang R. Gao, McGill University, Canada
Vivek Sarkar, IBM Software Solutions Division

This work studies the locality optimization problem for symmetric shared-memory multiprocessors (SMPs), a class of parallel machines that has experienced steady and rapid growth in the past few years.

We assume a distributed shared memory (DSM) model in which a global address, G, is mapped to (P,L), where P is the processor number, and L is the local memory address. In principle, G can be either a virtual or physical address, so can L. However, in this work, we will focus on the cases where either both G and L are physical (for physical addressed caches), or both G and L are virtual (for virtual addressed caches). We also assume that L is defined as a simple bitfield extraction from G.

The data layout of a variable specifies a mapping from G to (P,L) which consists of an alignment mask, AM (bit-value constraints on variable's starting address) and a processor mask, PM (bit positions to specify how P is extracted from G). Different PM values can be used to emulate different block-cyclic data layouts (with block sizes that are powers of 2). The mapping from G to (P,L) is assumed to be unique and identical for all aliased variables that map to G.

The program model considered in this work is a perfect loop nest, transformed by a sequence of iteration-reordering loop transformations such as permutation, reversal, skewing, tiling, parallelization, and coalescing. There is no restriction on the control flow and the variable references that can occur within the loop body.

The main focus of this work is an estimation of the memory performance of the loop nest for a given set of data layouts and loop transformations. We discuss how to estimate the number of cache misses arising from stale data invalidations and from false sharing in this DSM model, in addition to the cache misses arising from the usual uniprocessor considerations (cold misses, conflict misses, capacity misses). The goal of our work is to use this performance estimation to guide the selection of data layouts and loop transformations. We will also discuss the issues and challenges involved in automatically deriving the data layout parameters and loop tiling parameters from this performance estimation.

Static Estimation of Communication Costs of Data-Parallel Programs

Manish Gupta, IBM T.J. Watson Research Center
Prithviraj Banerjee, University of Illinois at Urbana-Champaign

We present a methodology to estimate communication costs of a program written in global address space, to guide data partitioning decisions. The analysis is performed by the compiler without actually generating communication, yet our approach attempts to capture the effect of important communication optimizations. We show that estimation based on analyzing just the volume and/or frequency of communication is often insufficient, there is also a need to take into account the parallelism or sequentiality amongst different parts of overall communication for a data reference. We describe a simple approach to characterize the nature and cost of communication for individual references at a high level, based on identification of some traversal properties of references over a processor grid. Finally, we present some experiences with using this methodology in an automatic data partitioning system.

Compile-Time Estimation of Communication Costs of Programs *

Manish Gupta† Prith Banerjee‡

Abstract

One of the most challenging problems in compiling for distributed memory machines is to determine how data for a program should be distributed across processors. Any compiler that makes data partitioning decisions needs a mechanism for estimating communication and computational costs of programs to compare different alternatives. This paper presents a methodology for estimating communication costs of programs written in global address space. In this approach, the compiler analyzes programs *before* generating communication, and yet takes into account important communication optimizations that will be performed. We introduce the notion of traversal properties of array references in loops, that help identify the nature and extent of data movement in terms of high-level communication primitives. This enables the compiler to obtain more precise information about the global state of communication, and in a largely machine-independent manner. The methodology described in this paper has been implemented in a compiler, PARADIGM, that automatically determines data partitioning for Fortran programs. The results obtained with PARADIGM confirm the importance of this analysis for making good data partitioning decisions. The techniques developed in this work for recognizing communication primitives that best characterize the data movement are quite general, and also form the basis of generation of communication in PTRAN-II, a prototype compiler for High Performance Fortran.

1 Introduction

Distributed-memory architectures have become extremely popular as a cost-effective method of building massively parallel computers. However, these machines are not very easy to program. The programmer has to perform low-level tasks like distributing data across processors, and managing communication among those processors. Languages like High Performance Fortran (HPF) [8] free the programmer from the burden of explicit message-passing by allowing him/her to write sequential or shared-memory parallel programs, annotated with directives specifying data decomposition. The compilers for these languages are responsible for partitioning the computation, and generating the communication necessary to fetch values of non-local data referenced by a processor. A number of such prototype compilers have been developed [13, 31, 17, 23, 16, 21, 12, 26].

The performance of these programs depends greatly on the data partitioning scheme chosen by the programmer. In general, the best partitioning scheme depends not only on program characteristics, but also on numerous machine-specific parameters, and on the kind of optimizations performed by the compiler. While early distributed memory compilers provided no assistance to the programmer in this crucial task, a number of researchers have recently addressed the problem (or parts of the problem) of automatically determining the distribution of data [2, 4, 7, 9, 11, 14, 15, 17, 20, 22, 28].

*This paper has appeared in the Journal of Programming Language, September 1994.

†IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598. Email: mgupta@watson.ibm.com. This work was done while the author was at the University of Illinois.

‡Center for Reliable and High Performance Computing, University of Illinois, Urbana, IL 61801. Email: banerjee@crhc.uiuc.edu

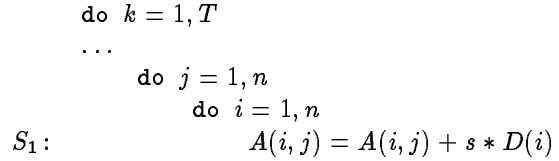


Figure 1: Effect of dependence on communication cost

Any strategy for automatic data partitioning needs a mechanism for cost estimation to compare different alternatives regarding data partitioning, and to evaluate the relative benefits and overheads of repartitioning data at various points. Most of the proposed approaches to automatic data distribution use very simplified metrics for communication costs. The resultant inaccuracies in the estimated cost can often prevent the compiler from reaching good data partitioning decisions, even if the underlying algorithms for the selection process are good. Based on our experience with PARADIGM [11, 9], a compiler that automatically determines data distribution for Fortran programs, we have found the following features to be extremely important for any performance estimator guiding data partitioning decisions:

- *Incorporating data dependence information*: The *flow dependences* [30] to an array reference being read in a statement inside loops restrict the extent to which messages for that reference can be combined [31, 13]. For example, in Figure 1, the flow dependences involving the reference to $D(i)$ will govern the placement of its communication. If there is a dependence *carried* by the i-loop, there is a separate message needed to send the value of $D(i)$ to the processor executing statement S_1 (if it is different from the processor owning $D(i)$), during each iteration of the loops surrounding S_1. At the other extreme, if there is no flow dependence to that reference, all communication involving the array D can be done once before entering the k-loop. Thus, communication would be aggregated with respect to the two inner loops, and its redundant repetitions in the k-loop would be eliminated. Given the high cost of communication and the significant difference between startup costs and per-element costs of sending messages, it is clear that the incurred costs will differ greatly in these cases. Thus, the estimated cost should take into account dependence information, otherwise it can be considerably inaccurate.
- *Providing a high-level view of communication*: Even if the compiler generates communication using only `send` and `receive` primitives, it is useful to recognize the overall data movement taking place across processors in terms of higher-level primitives. For instance, consider the different communication patterns shown in Figure 2, each consisting of three `send-receives` of identical message sizes, and with no constraints on their ordering. Even though the number of `send-receives` is exactly the same in these cases, the overall communication cost for these patterns will be different. In this example, we expect $\text{cost}(C_3) < \text{cost}(C_1) < \text{cost}(C_2)$. Characterization of data movement in terms of collective communication primitives leads to a concise and machine-independent representation that is suitable for data-parallel programs. This representation also provides richer information about the global state of communication (for instance, whether it is a one-to-many, many-to-one, or a many-to-many communication pattern), that can be useful for modeling effects like contention in the network.

In this paper, we present a methodology for estimating communication costs, that analyzes the program before communication is generated by the compiler. Our approach for estimating computational costs is described elsewhere [9]. The estimates of communication costs are expressed in terms of times to carry out certain high-level communication primitives on the target machine. Given information regarding the performance characteristics of those primitives on a multicomputer, the actual times spent on communication can then be estimated for that specific machine. This methodology has been implemented in the PARADIGM compiler [9]. The analysis we have developed introduces new techniques for determining how data movement

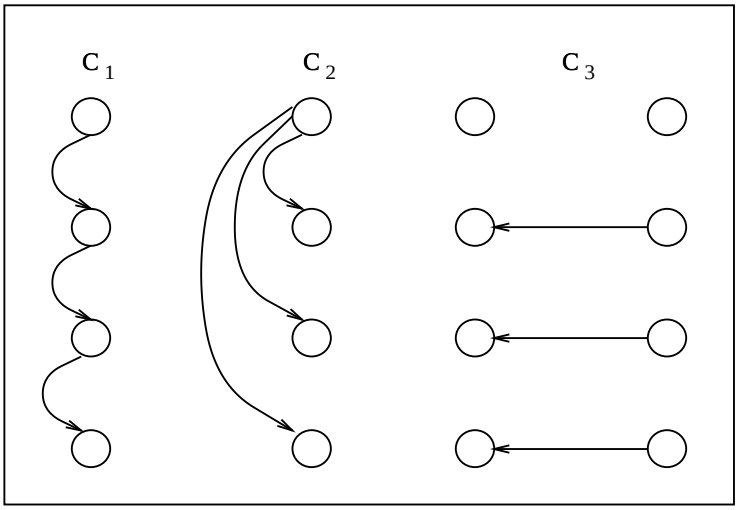


Figure 2: Need for high-level view of communication

needed in a program can be implemented efficiently using collective communication primitives, and forms the basis of generation of communication by the PTRAN II compiler [12].

Related Work Many researchers have developed tools [3, 5, 6, 27] that estimate the performance of a parallel program with explicit communication at compile time. The tool developed by Balasundaram et al. [3] employs the "training sets" approach to determine performance characteristics of collective communication routines, and then uses that information to predict performance of the parallel program. Fahringer's *Parameter based Performance Prediction Tool* [5, 6] uses profiling to obtain sequential program parameters, and then determines parallel program parameters that include measurements on load distribution, communication overhead, and data locality. The availability of detailed information about the generation of communication enhances the accuracy of cost estimation. However, a drawback of using these tools to guide data partitioning decisions is that the compiler has to generate the SPMD program with explicit communication for *each* data partitioning scheme to be evaluated. This can be prohibitively expensive.

A number of other researchers also analyze the source program without explicit message-passing to guide data partitioning decisions. Anderson and Lam [2] recognize nearest-neighbor shifts, data-reorganization communication, and pipelined communication. They use qualitative distinctions between the cost of these primitives while determining data and computation partitions. Chatterjee et al. [4] use cost measures representing the volume of communication, and ignore the effect of optimizations like aggregation of communication. In contrast to these approaches, we obtain more detailed estimates in terms of the size of data and the number of processors involved in each communication in the program. We show that this level of detail is often required for data partitioning decisions, particularly those determining the number of processors on which different array dimensions should be distributed.

Wholey [28] uses cost functions for various communication primitives on the target machine to estimate communication overheads for programs written in the ALEXI language. He deals with a simpler problem, since the ALEXI programs already have calls to high level primitives that correspond closely to collective communication operations.

Our research extends the work of Li and Chen [17], who use pattern-matching on array references to generate high-level collective communication. They deal with a simpler, single-assignment language, and their techniques are suited to a relatively restricted class of references. For instance, they do not handle references with coupled subscripts very well. We have introduced the notion of synchronous properties between array references, that enables the compiler to perform a more sophisticated analysis.

Organization of the Paper The rest of the paper is organized as follows. Section 2 presents a framework for determining some basic properties of data references, that help characterize the data movement involving multiple instances of those references. Section 3 describes how the compiler determines the placement of communication for a reference, that also governs the extent to which communication for multiple instances of that reference can be aggregated. Section 4 describes how the overall cost of communication for a reference is determined. Section 5 presents experimental results on automatic data partitioning for some programs, that show the importance of using good performance estimates to guide those decisions. Finally, Section 6 presents conclusions and ideas for future work.

2 Characterization of Data Movement

In this section, we describe properties of array references, which characterize the nature and the extent of data movement that they are involved in. We first describe how the arrays may be distributed, and how the cost of data movement is expressed. While in this paper we shall describe the estimation of communication costs only for arrays, our techniques are also applicable to communication involving scalar variables. Most compilers either replicate or privatize scalar variables. The PTRAN II compiler chooses between replication and *non-replicated alignment* of a scalar with an array reference [12], that covers privatization and mapping to a single processor as special cases. In either case, the distribution of scalars is subsumed by the possible distributions that we consider for arrays. Therefore, our techniques for characterizing communication involving arrays can be used for scalars as well.

2.1 Distribution of Arrays

In languages like HPF [8], the mapping of arrays is specified at two levels: arrays are aligned with templates or other arrays, those alignment targets are further distributed onto processors. Eventually, each array dimension is either collapsed or distributed over a processor grid dimension via (i) partitioning in a block, cyclic, or block-cyclic manner, (ii) replication, or (iii) mapping to a constant processor position [12]. We do not initially consider replication or mapping of an array dimension to a constant processor position. In Section 4.1.6, we describe how the results for these cases are obtained as straightforward extensions to the analysis shown in the rest of the paper. Given the form of alignment and distribution directives allowed by HPF, the mapping function of a partitioned array dimension, A_k (the kth dimension of array A), can be expressed as:

$$f(A_k, i) = \left\lfloor \frac{s_k * i - o_k}{b_k} \right\rfloor [\mathrm{mod} N_K]$$

This function returns the processor position to which the ith element along the array dimension is mapped. The term modN_K is needed only for cyclic and block-cyclic distributions, that we shall collectively refer to as *general cyclic* distributions. The parameters s_k and o_k are referred to as the stride and offset of the distribution, while b_k is referred to as the block size.

2.2 Communication Primitives

The compiler estimates communication costs in terms of the time taken to carry out the following primitives. These primitives are similar to the collective communication primitives supported by the Message Passing Interface [19], an emerging standard for message passing on parallel machines.

- `Transfer or Send-Receive` : a single source processor sends a message to a single destination processor.
- `OneToManyMulticast` : a single source processor sends a message to all other processors in a group.

Primitive	*Cost*
Reduction(m, p)	$\lceil \log_2(p) \rceil *$ Transfer(m)
Shift(m)	$2*$ Transfer(m)
OneToManyMulticast(m, p)	$\lceil \log_2(p) \rceil *$ Transfer(m)
ManyToManyMulticast(m, p)	$(p-1)*$ Shift(m)
Scatter(m, p)	$(p-1)*$ Transfer(m)
Gather(m, p)	$(p-1)*$ Transfer(m)

Table 1: Cost functions of collective communication primitives under a simple implementation

- `ManyToManyMulticast` : all processors in the group send data to all other processors in a group.
- `Scatter` : a single source processor sends *different* messages to all other processors in a group.
- `Gather` : all processors send messages to a single destination processor in a group.
- `Shift` : circular shift of data among adjacent processors in a group.
- `Reduction` : reduction of data using a simple associative and commutative operator, over all of the processors in a group.

Figure 3 illustrates the data movement associated with each routine. These primitives serve as a convenient form for representing communication costs, since they capture data movement in a fairly detailed, yet machine-independent manner. Even if these primitives (other than Send-Receive) are not supported, or are not used by the compiler in generating communication, information conveyed by their representation about the one-to-many, many-to-one, or many-to-many form of communication leads to better precision in global cost estimation.

The cost of a Transfer (Send-Receive) operation on m bytes for most distributed memory machines can be approximated as:

$$\text{Transfer}(m) = \nu + \lambda * m,$$

where ν and λ are, respectively, the start-up cost and the per-element cost of sending a message, and they may sometimes take different values depending on the message length. In this paper, we ignore the effect of distance between the processors on communication cost. This assumption is quite reasonable for most modern machines. For simplicity, we also ignore the overlap between communication and computation, and the buffer copying overhead associated with sending non-contiguous data.

The cost of collective communication operations can vary considerably in form across different implementations. Table 1 lists the cost functions used in PARADIGM corresponding to an implementation using Transfer operations, described below. The parameter p denotes the number of processors over which the primitive is carried out. Both Reduction and OneToManyMulticast take $\log_2(p)$ steps with tree-based algorithms. A Shift operation involves each processor sending data to its neighboring processor, and receiving data from another neighboring processor in the other direction, and hence is modeled as taking time equivalent to two Transfers. A ManyToManyMulticast over a group of p processors is done via $p-1$ Shifts using a ring-based algorithm. A Scatter/Gather operation is implemented as a sequence of Transfers, where the source/destination processor sends/receives different pieces of data to/from the $p-1$ other processors. These cost measures can be modified appropriately when the algorithms used are different.

2.3 Characterization of Array References

PARADIGM analyzes the expression corresponding to each subscript, and assigns it to one of the following types: `constant`, `single-index` (affine function of a single loop induction variable), `multiple-index`

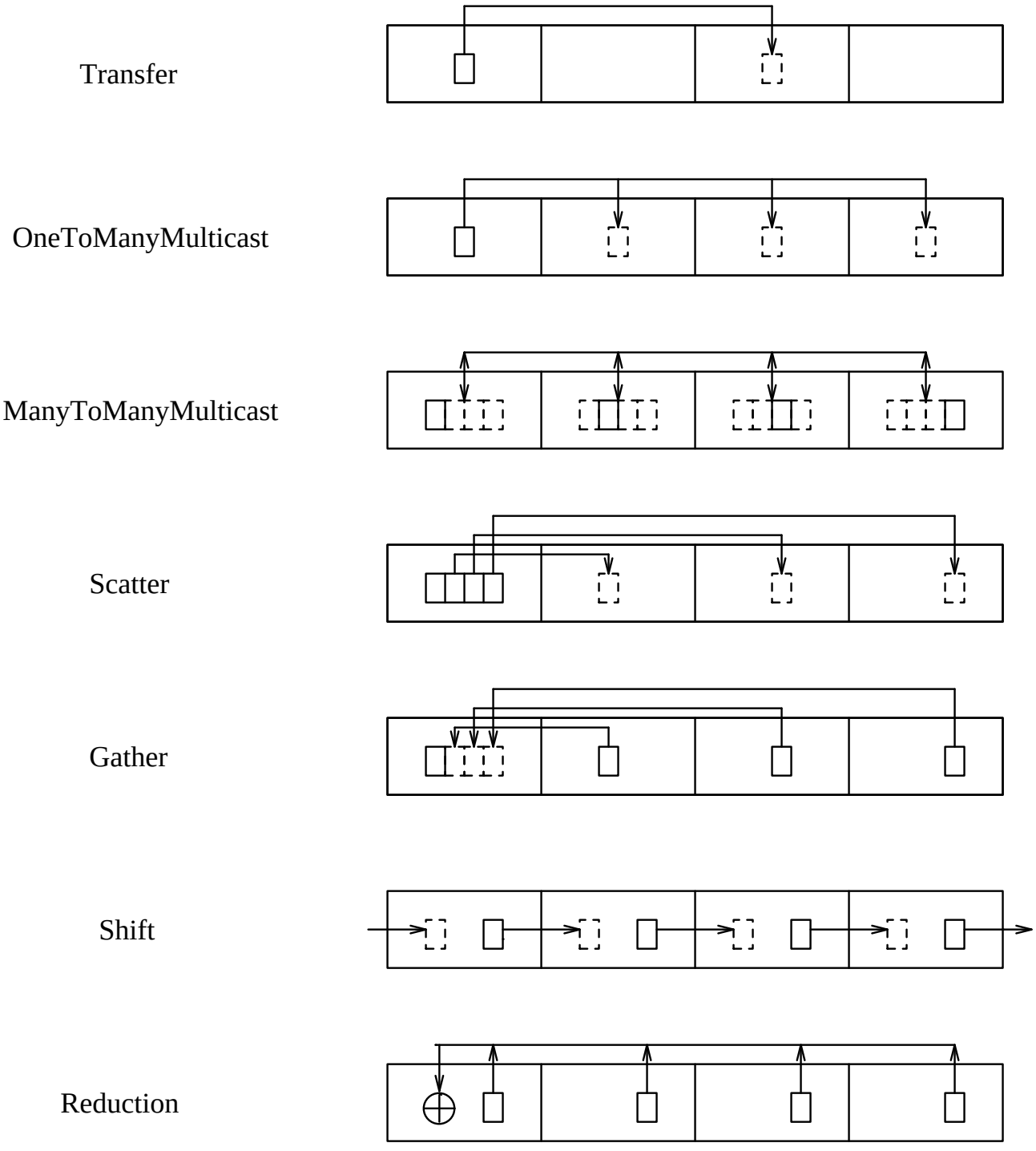


Figure 3: Communication primitives recognized by PARADIGM

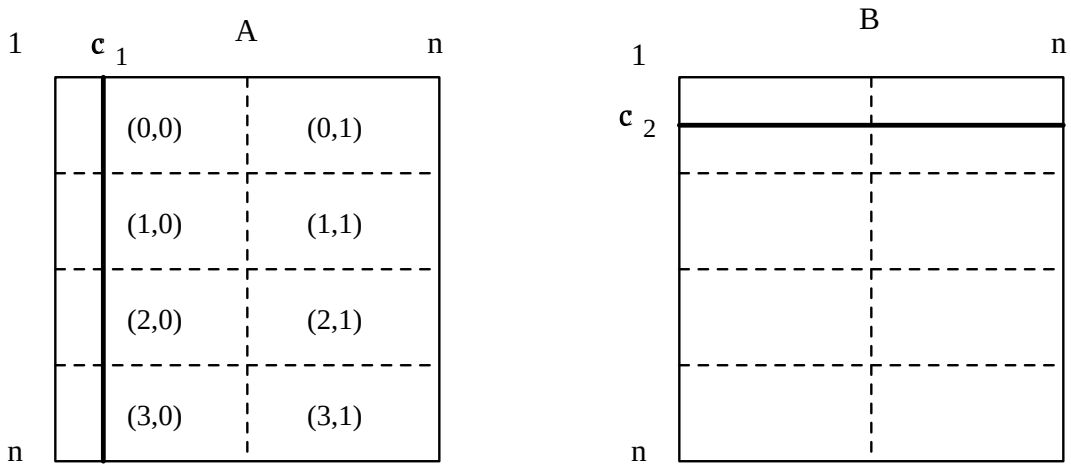


Figure 4: Traversal of sub-references in a loop

(affine function of more than one loop induction variable), or `complex`. Information is also kept about its `variation-level`, i.e., the innermost loop in which the subscript changes values. In order to simplify cost estimation, all loops are normalized prior to the analysis.

We refer to a subscript associated with a distributed dimension in an array reference as a *sub-reference*. A sub-reference can be represented by a tuple $\langle r, d \rangle$, where r is an array reference, and d is the position ($d \geq 1$) of the subscript in that reference. A sub-reference varying inside a loop can be viewed as traversing a sequence of elements distributed on different processors along a grid dimension. Figure 4 shows the traversal of two sub-references, $\langle A(i, c_1), 1 \rangle$ and $\langle B(c_2, i), 2 \rangle$, by solid lines. The set of processor positions in a grid dimension traversed by a sub-reference sr in a loop L is referred to as its *processor position set* with respect to the loop, denoted as $\texttt{PPS}(sr, L)$. Similarly, the set of subscript positions along the array dimension traversed on the memory of a processor at position p is referred to as its *subscript position set*, denoted as $\texttt{SPS}(sr, L, p)$.

The number of processor positions traversed by the sub-reference, i.e., the size of its processor position set is referred to as $\texttt{NPP}(sr, L)$. The number of subscript positions traversed on a particular processor position is referred to as $\texttt{NSP}(sr, L, p)$. Another useful quantity to compute is the maximum size of any subscript position set of a sub-reference with respect to a loop, denoted as $\texttt{MNSP}(sr, L)$. It is defined as the maximum value of $\texttt{NSP}(sr, L, p)$ for all p belonging to $\texttt{PPS}(sr, L)$. For the sub-reference $sr = \langle A(i, c_1), 1 \rangle$ in Figure 4, it can be seen that $\texttt{NPP}(sr, L) = 4$, and $\texttt{MNSP}(sr, L) = \lceil n/4 \rceil$.

Each point in the loop (identified by the value of loop index) at which the sub-reference crosses a processor boundary in that grid dimension is called a *transition point* of the loop for that sub-reference. We now define some properties describing the relationships between sub-references varying inside the same loop, that help characterize the data movement for that loop. That, in turn, helps identify the communication primitive that is suitable for implementing the data movement.

k-Synchronous Property A sub-reference sr_1 is said to be *k-synchronous* (k is a positive integer) with another sub-reference sr_2, with respect to a loop L, if (i) every transition point of L for sr_2 coincides with a transition point of L for sr_1, and (ii) between every two consecutive transition points of L for sr_2, there are exactly $k - 1$ transition points of L for sr_1.
Example: In Figure 4, the sub-reference $\langle A(i, c_1), 1 \rangle$ is 2-synchronous with $\langle B(c_2, i), 2 \rangle$, with respect to the i-loop.

Strictly Synchronous Property A sub-references sr_1 is said to be *strictly synchronous* with another sub-reference sr_2, with respect to a loop L, if both sub-references are mapped to the same processor position during each iteration of L. We observe that this implies that (i) sr_1 is 1-synchronous with sr_2, with respect to L (i.e., every transition point of L for sr_1 is also a transition point of L for sr_2, and vice versa), and (ii) the coinciding transition points represent the cross-over points between the same processor positions in the respective grid dimensions for those sub-references.
Example: In Figure 4, if B_2 were distributed on four processors (rather than two), the sub-reference $\langle A(i, c_1), 1\rangle$ would be strictly synchronous with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop.

A useful convention adopted regarding the k-synchronous property is that k can also be a reciprocal of a positive integer. In that case, a statement that sr_1 is k-synchronous with sr_2 is interpreted as conveying that sr_2 is $1/k$-synchronous with sr_1.

2.4 Static Determination of Characteristics

Given information about the distribution of arrays, we now show how the compiler determines the above characteristics of sub-references. All of the results presented in this section are for sub-references of the type `single-index`. The corresponding analysis for sub-references of the type `multiple-index` is based on these results, and for sub-references of the type `complex` cannot be done precisely at compile-time. We present results for the general case of block-cyclic distribution, of which block and cyclic distributions are special cases.

2.4.1 Traversal of Sub-references

Consider a sub-reference sr of the type `single-index`, with subscript expression $e = \alpha * j_l + \beta$, varying in a (normalized) loop L_l (loop at level l) with a range of r_l. Let α' denote $|\alpha|$, and let the corresponding array dimension A_k have a distribution function shown in Equation 1. The number of processor positions traversed by sr in L_l is:

$$\texttt{NPP}(sr, L_l) \quad = \quad \min(N_k, \lceil (s_k \alpha' r_l)/b_k \rceil) \tag{1}$$

The second term in the above expression determines the number of data blocks traversed by the sub-reference, which also gives the number of processors traversed, in the case of block distribution. For general cyclic distributions, the above expression takes into account the wrap-around of data blocks over N_k processors.

The maximum number of subscript positions, over any processor, traversed by sr in L_l is estimated as:

$$\begin{aligned} \texttt{MNSP}(sr, L_l) \quad = \quad & (\lfloor b_k/(s_k\alpha') \rfloor * \lfloor (s_k \alpha' r_l)/(N_k b_k) \rfloor) + \\ & \min(\lfloor b_k/(s_k\alpha') \rfloor, \lceil ((s_k \alpha' r_l) \bmod (N_k b_k))/(s_k \alpha') \rceil) \end{aligned} \tag{2}$$

The traversal of the sub-reference conceptually consists of two regions: region P, which is equally distributed among processors, and region Q, which covers only some of the processors. The term $\lfloor (s_k \alpha' r_l)/(N_k b_k) \rfloor$ denotes the number of complete blocks covered in region P on each processor. The term $\lfloor b_k/(s_k\alpha') \rfloor$ denotes the number of subscript positions visited in a single block of A_k on a processor. Thus, the product of these two terms (the first additive term in the above expression) represents the contribution corresponding to the region P. If the extent of region Q exceeds the block size of A_k, the maximum number of additional subscript positions mapped to a processor is $\lfloor b_k/(s_k\alpha') \rfloor$. Otherwise, that additional number is $\lceil ((s_k \alpha' r_l) \bmod (N_k b_k))/(s_k \alpha') \rceil$, i.e., the total number of subscript positions corresponding to the region Q.

2.4.2 Synchronous Properties

We now present tests to detect the synchronous properties for a pair of sub-references. The corresponding array dimensions must be distributed in the same manner – block or general cyclic, and in case of general

cyclic distribution, on an equal number of processors. Let the subscript expressions of the two sub-references, sr_1 and sr_2, be $e_1 = \alpha_1 * i + \beta_1$, and $e_2 = \alpha_2 * i + \beta_2$. Let b_1 and b_2 be the block sizes, s_1 and s_2 be the strides, and o_1 and o_2 be the offsets of distribution of the corresponding array dimensions.

Strictly synchronous sub-references The following theorem gives sufficient conditions for the strictly synchronous property.

Theorem 1 *Let sr_1 and sr_2 be two sub-references, as described above. The sub-reference sr_1 is strictly synchronous with sr_2, with respect to loop L if either of the following sets of conditions are satisfied:*

- *(i) $s_1\alpha_1/b_1 = s_2\alpha_2/b_2$, and (ii) $(s_1\beta_1 - o_1)/b_1 = (s_2\beta_2 - o_2)/b_2$,*
  or
- *(i) $b_1 = ms_1\alpha_1$, (ii) $b_2 = ms_2\alpha_2$, and (iii) $\lfloor (s_1\beta_1 - o_1)/s_1\alpha_1 \rfloor = \lfloor (s_2\beta_2 - o_2)/s_2\alpha_2 \rfloor$,*
  where m is a positive integer.

Proof Given in the appendix.

k-synchronous sub-references The conditions to check whether sr_1 is k-synchronous with sr_2 are obtained in a similar manner, and are shown below:

Theorem 2 *Let sr_1 and sr_2 be two sub-references as described above. The sub-reference sr_1 is k-synchronous (k being an integer) with sr_2, with respect to loop L if either of the following sets of conditions are satisfied:*

- *(i) $s_1\alpha_1/b_1 = k * (s_2\alpha_2/b_2)$, and (ii) $(s_1\beta_1 - o_1)/b_1 = k * ((s_2\beta_2 - o_2)/b_2) + l$, where l is an integer,*
  or
- *(i) $b_1 = ms_1\alpha_1$, (ii) $b_2 = k * (ms_2\alpha_2)$, and (iii) $\lfloor (s_1\beta_1 - o_1)/s_1\alpha_1 \rfloor = \lfloor (s_2\beta_2 - o_2)/s_2\alpha_2 \rfloor + ml$,*
  where m and l are integers, and $m > 0$.

Proof Omitted, similar to the proof of Theorem 1.

2.4.3 Other Methods of Characterization

During the analysis of communication requirements, if the compiler is unable to establish a k-synchronous property between two sub-references of the type `single-index`, it uses one of the following two measures to characterize the relationship between the two sub-references. The first measure is the *speed-ratio*, defined as:

$$\texttt{SpRatio}(sr_1, sr_2, L) \quad = \quad \lceil (s_1\alpha_1 b_2)/(s_2\alpha_2 b_1) \rceil \tag{3}$$

This captures roughly, the "speed" with which the sub-reference sr_1 crosses processor boundaries in the loop L, relative to the sub-reference sr_2. This measure is a generalization of the notion of k-synchronous property, and it may be observed that a k-synchronous property between two sub-references for a given loop implies a `speed-ratio` of k.

Boundary-communication The "boundary-communication" is a specialized test performed between sub-references corresponding to aligned dimensions, and related by a `speed-ratio` of one. This test helps detect data movement taking place across boundaries of regions allocated to neighboring processors. It checks for the following conditions:

```
do j_1 = 1, r_1
  . . .
      do j_l = 1, r_l
         ⟨ communication for B(h_1, ..., h_q) ⟩
         do j_{l+1} = 1, r_{l+1}
            . . .
                 do j_m = 1, r_m
                      A(g_1, g_2, ..., g_p) = F(B(h_1, h_2, ..., h_q))
```

Figure 5: Statement requiring communication of data

1. $s_1\alpha_1/b_1 = s_2\alpha_2/b_2$.
2. $|(s_1\beta_1 - o_1)/b_1 - (s_2\beta_2 - o_2)/b_2| \leq 1$.

If the above conditions are satisfied, the amount of data transfer across the boundary of each participating processor is estimated as:

$$\texttt{BD}(sr_1, sr_2, L) = (\lceil |(s_1\beta_1 - o_1)/(s_1\alpha_1) - (s_2\beta_2 - o_2)/(s_2\alpha_2)| \rceil) * (\lceil (s_2\alpha_2 r_l)/(N_k b_2) \rceil) \quad (4)$$

The first term, $\lceil |(s_1\beta_1 - o_1)/(s_1\alpha_1) - (s_2\beta_2 - o_2)/(s_2\alpha_2)| \rceil$ gives the number of elements transferred across a single block of the array dimension. The second term in the product gives the number of blocks (among those traversed by the sub-reference sr_2) that are held by a single processor.

2.4.4 Handling Statically Unknown Parameters

The analysis described in this section assumed a complete knowledge of program parameters, such as loop bounds, sizes of arrays and their distribution parameters. Some of the analysis can be performed symbolically, or by reformulating the results differently. For instance, consider the test for strictly synchronous property between a pair of sub-references. Even if the block sizes are not known statically due to the number of processors being unknown, it may be known that the two block sizes are the same because of the arrays having identical shapes. The first set of conditions in Theorem 1 would simplify to $s_1\alpha_1 = s_2\alpha_2$ and $s_1\beta_1 - o_1 = s_2\beta_2 - o_2$. Similarly, the test for b_1 being an integer multiple of $s_1\alpha_1$ can be structured so as to first check for the commonly occurring case of $s_1 = \alpha_1 = 1$, in which case the test is satisfied trivially.

For program parameters that are essential for determining the traversal properties, PARADIGM queries the user for values to be used in its estimates. This method was chosen primarily for the ease of implementation. A more advanced method for determining unknown parameters such as loop bounds would be to use profiling [25, 6]. However, profiling is sometimes not feasible because of the time required to run the program sequentially for meaningful data sizes.

3 Placement of Communication

Consider a reference appearing in a statement inside one or more loops, as shown in Figure 5. If possible, the communication for that reference should be moved outside of loops to enable *message vectorization*, which combines a sequence of messages on individual array elements into a single message [13, 31]. The compiler examines all incoming flow dependences to the reference, and the innermost loop carrying any of those dependences identifies the loop from which communication cannot legally be moved outside.

```
CommunicationLevel(R, S)
{
   lmax = 0
   for each dependence into S do
       if ((dependence-type == flow) and (dependence due to reference R))
          S' = source statement of dependence
          determine k = nesting level of dependence
          for (l = k;  l ≥ 1;  l--) do
              if S and S' belong to same strongly connected component in dependence graph
              at level l
                  break
              endif
          endfor
          lmax = max(l, lmax)
       endif
   endfor
   Identify the doall loops outside loop L_lmax as loops to be brought inwards.
}
```

Figure 6: Algorithm to determine communication level for reference R in statement S

3.1 Loop Transformations for Combining Messages

Prior to determining the placement of communication while estimating costs, PARADIGM considers loop transformations that expose more opportunities for combining communication. The application of transformations is a compiler-dependent feature, and PARADIGM tries to incorporate some important optimizations that have been mentioned in the literature [31, 13, 2, 12]. When used along with a compiler that performs/does not perform certain transformations, the estimation algorithm can be modified appropriately. While estimating communication costs, PARADIGM does not consider transformations for improving data locality, such as loop fusion or loop interchange that is geared towards better cache performance. These transformations have a more significant impact on computational costs, and can usually be accomodated by first moving communication outside using the transformations we shall describe, and then transforming the inner loops for better locality [12].

It is often useful to defer the actual application of transformations for communication or even the decision to apply them, until data partitioning decisions are made, since their utility may itself depend on data partitioning. Hence, we have simply incorporated legality checks for these transformations in the algorithm for determining placement of communication in the cost estimator of PARADIGM [9]. The algorithm is described in Figure 6.

Loop Distribution The distribution of a loop over two components enables any communication placed between those components to be aggregated with respect to that loop. For example, in the program segment shown in Figure 7, distributing the i-loop allows the communication involving D to be moved outside that loop. The algorithm shown in Figure 6 examines the legality of loop distribution by checking if the source and the target of a loop carried dependence belong to different strongly connected components in the dependence graph at that level [1].

Loop Permutations Whenever there is a parallel loop outside a loop in which communication takes place, the parallel loop should be brought inwards to ensure that communication is aggregated with respect to that loop. A loop permutation that brings a parallel loop inwards is always legal, and it cannot adversely affect the combining of communication for any other reference. In the program segment shown in Figure 8,

```
do i = 1, n
     D(i) = D(i) + s * B(i)
     D(i) → owner(A(i,1 : n))
     do j = 1, n
          A(i, j) = A(i, j) + D(i)
     enddo
enddo
```

```
do i = 1, n
     D(i) = D(i) + s * B(i)
enddo
D(i) → owner(A(i,1 : n)),i = 1 : n
do i = 1, n
     do j = 1, n
          A(i, j) = A(i, j) + D(i)
     enddo
enddo
```

Figure 7: Loop distribution

```
do i = 1, n
     do j = 1, n
          A(i, g(j)) → owner(A(i,j))
          A(i, j) = F[A(i, g(j))]
     enddo
enddo
```

```
do j = 1, n
     A(i, g(j)) → owner(A(i,j)),i = 1 : n
     do i = 1, n
          A(i, j) = F[A(i, g(j))]
     enddo
enddo
```

Figure 8: Loop interchange

communication for the `rhs` reference cannot be taken outside the j-loop, because of the potential flow dependence carried by that loop. Following loop interchange, which brings the outer parallel loop inwards, that communication can be vectorized.

In the above example, it is important to recognize the possibility of loop interchange while estimating communication cost when A_2 is partitioned across processors, even though loop interchange will not be needed if A_2 were collapsed in the finally chosen data partitioning scheme. Thus, it is useful to have the ability to account for the benefits of loop transformations that will be performed, without actually transforming the program.

3.2 Limits on Combining of Messages

The algorithm shown in Figure 6 determines the outermost loop level at which communication can be placed. However, the compiler may need to control the extent to which messages are combined, for instance, to limit the size of communication buffers required. This can be achieved by stripmining loops, as shown in Figure 9. On stripmining a loop with strip size b, the number of messages increases by a factor of b, but the total size of data communicated remains the same. Since stripmining is (or should be) done at large message sizes, when the start-up costs account for only a small fraction of the overall cost of sending messages, the overall effect of stripmining on communication costs can be reasonably ignored.

Another consequence of the above algorithm is that potentially, the compiler may generate more communication than necessary. Consider the program segment shown in Figure 10. The communication for the reference to $D(i)$ would be placed outside the j-loop, and communication would be realized for all instances of that reference, even though the assignment statement may not be executed for all the loop iterations. Information about the frequency of execution of statements, possibly from profiling, can help the compiler decide between carrying out potentially extra communication and using a larger number of messages. However, we do not know of any compiler that examines this trade-off while generating communication. PARADIGM

```
do j = 1, n
   A(i, g(j)) → owner(A(i,j)),i = 1 : n
   do i = 1, n
      A(i, j) = F[A(i, g(j))]
   enddo
enddo
```

```
do j = 1, n
   do i' = 1, n, b
      A(i, g(j)) → owner(A(i,j)),i = i' : i' + b − 1
      do i = i', min(i' + b − 1, n)
         A(i, j) = F[A(i, g(j))]
      enddo
   enddo
enddo
```

Figure 9: Loop stripmining

```
do j = 1, n
   do i = 1, n
      if (s ≠ 0) then
         A(i, j) = D(i)/s
      end if
      ...
```

Figure 10: Extra communication in the presence of conditionals

assumes that communication will be taken outside loops even in the presence of conditionals inside them.

4 Estimation of Communication Cost

This section describes how the overall cost of communication for a reference is estimated, after the compiler has determined the outermost level at which communication can be placed, as shown in Figure 5. We refer to the inner $m - l$ loops, with respect to which communication can be aggregated, as Type-1 loops, and to the outer l loops as Type-2 loops. A high-level overview of the algorithm for estimating cost for a reference, R, is given in Figure 11. The reference, L, refers to the data whose owner is assigned the computation corresponding to the statement. Usually, L is set to the left hand side (`lhs`) reference for an assignment statement, in accordance with the *owner computes* rule [13]. It is chosen differently for conditional statements, and when the compiler overrides the owner computes rule [12]. The basic idea of the algorithm in Figure 11 is to obtain a cost estimate of the sequence of communication primitives that implements the data movement required for the entire iteration space of Type-1 loops, and multiply it by the expected number of times those primitives would be executed in the program. The cost estimate of the sequence of primitives is derived by first obtaining cost terms corresponding to "one-dimensional" views of the data movement in each processor grid dimension, and then composing those terms together.

The first step of our algorithm matches pairs of sub-references corresponding to aligned array dimension. In that step, if the `lhs` and the `rhs` arrays differ in the number of dimensions, the array with fewer dimensions is augmented with dummy, "missing" dimensions, that match extra distributed dimensions of the other array. A "missing" sub-reference is regarded as one of the type `constant`.

We now describe how communication cost terms are obtained for aggregate data movement corresponding to sub-references varying in a Type-1 loop, and for single data movement corresponding to other sub-references. We then show how these terms are composed to give an overall estimate of the communication cost.

```
CommunicationCostEstimate(R, L)
{
   Match pairs of sub-references of R and L corresponding to aligned array
   dimensions, and set variation-level of each pair to the maximum of
   variation-level values of the two sub-references.

   for (k = m;  k > l;  k-- ) do
      Obtain communication cost terms for all sub-reference pairs and for all rhs
      subscripts in collapsed dimensions, that have a variation-level of k [Section 4.1].

      Update the value of communication level, l, if indicated by the above step.
   endfor
   For each pair of sub-reference not yet considered, obtain communication cost term
   [Section 4.2].

   Compose all cost terms, and multiply them by count of repetitions [Section 4.3].
}
```

Figure 11: Algorithm to estimate cost of communicating data for reference R to owner of reference L

4.1 Cost Terms for Aggregate Data Movement

The data movements required in different iterations of a Type-1 loop can legally be combined. Therefore, the compiler attempts to recognize the communication primitive that best realizes the collective movement, and obtains the term(s) representing its cost. The techniques we have developed allow this analysis to be carried out for arbitrary kinds of sub-references, and for any arbitrary number of pairs of sub-references varying in a loop. For ease of presentation, we shall describe different aspects of the analysis separately.

4.1.1 Single pair of varying sub-references

Table 2 lists the communication term(s) obtained if there is a single pair of sub-references varying in such a loop L. This table enumerates the cases corresponding to only the "basic" categories of the subscripts for the `lhs` and `rhs` sub-references. The results for the other cases (when the two subscripts of the type `single-index` have different values of `variation-level`, or when one of the subscripts is of the type `multiple-index`) are derived in terms of these results, and are presented later in this section. The column marked *conditions tested* lists the tests performed by the compiler to obtain further information about the nature of data movement. The entry *reduction op* represents the test to see if the `rhs` reference is involved in a reduction operation (such as addition, max, min) inside the loop.

The case corresponding to both the sub-references being of the type `single-index` is perhaps the most commonly occurring pattern inside various loops in the scientific application programs. If the two sub-references satisfy one of the five tests shown in the table, it means that the processors can be partitioned into mutually disjoint groups over which the communication primitives indicated by the cost terms may be carried out in parallel. Figure 12 shows the different kinds of data movement corresponding to those cases. The term IDM(m) (internalized data movement of m elements) by itself does not represent any communication cost, it affects the data sizes of other communication cost terms during the composition step, as shall be discussed later. The third case corresponds to data movement that is partly intra-processor and partly inter-processor in the form of nearest-neighbor communication. Hence, it is represented by a Shift and an IDM term. In the fourth case, the symbol *D-Scatter* represents a Scatter operation over a "different" group, i.e., a group that does not include the source processor sending data values. A D-Scatter operation involves data being sent to p other processors, rather than $p-1$ (this distinction is important for accuracy

LHS (sr_1)	*RHS* (sr_2)	*Conditions Tested*	*Communication Term*
single-index	constant	default	OneToManyMulticast$(1, \texttt{NPP}(sr_1, L))$
constant	single-index	1. reduction op	Reduction$(1, \texttt{NPP}(sr_2, L))$
		2. default	Gather$(\texttt{MNSP}(sr_2, L), \texttt{NPP}(sr_2, L))$
single-index	single-index	1. sr_1 strictly synch sr_2	IDM$(\texttt{MNSP}(sr_2, L))$
		2. sr_1 1-synch sr_2	$(N_I > 1)*$ Transfer$(\texttt{MNSP}(sr_2, L))$
		3. boundary-commn.	(i) $(N_I > 1)*$ Shift$(\texttt{BD}(sr_1, sr_2, L))$
			(ii) IDM$(\texttt{MNSP}(sr_2, L) - \texttt{BD}(sr_1, sr_2, L))$
		4. sr_1 k-synch sr_2, $k > 1$	D-Scatter$(\texttt{MNSP}(sr_2, L)/k, k)$
		5. sr_1 k-synch sr_2, $k < 1$	D-Gather$(\texttt{MNSP}(sr_2, L), 1/k)$

Table 2: Communication terms for a pair of sub-references varying in a Type-1 loop

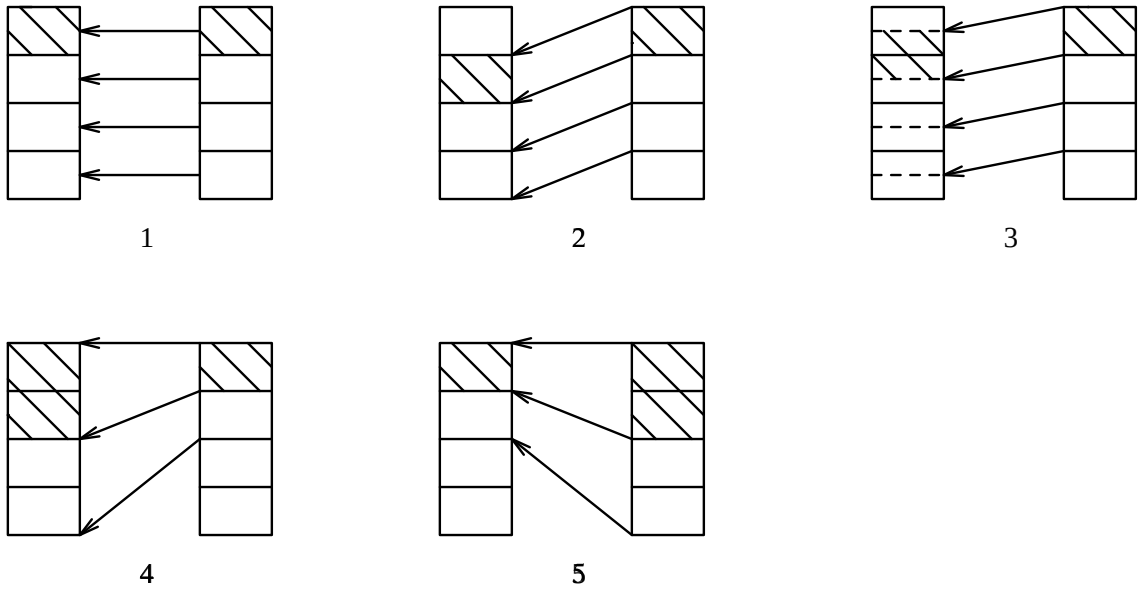


Figure 12: Data movements for sub-references of the type `single-index`

when p takes a small value). Thus, the term D-Scatter(m, p) is simply short-hand for the following two terms: (i) $(N_I > 1) *$ Transfer(m), and (ii) $(N_I > 1) *$ Scatter(m, p), where N_I is the number of processors on which the aligned array dimensions are distributed. The term D-Gather(m, p) is defined in a similar manner.

If none of the five tests indicated in the table are satisfied, the compiler computes the value of $\texttt{SpRatio}(sr_1, sr_2, L)$ using Equation 3. Similar to the last two entries for the k-synchronous property, a D-Scatter or a D-Gather term is used depending on the whether the value of `speed-ratio` is greater than or less than one. However, in this case, the Scatter or the Gather operations corresponding to different groups do not take place entirely in parallel, since there is an overlap of processors in those groups. This overlapping between groups, and the resulting contention during the process of communication is modeled by multiplying each of those terms (represented by D-Scatter or D-Gather) by a factor of two.

4.1.2 Multiple pairs of varying sub-references

When there is more than one pair of sub-references varying in a Type-1 loop, the analysis shown above is insufficient. For example, consider the program segment shown in Figure 13, where both the arrays A

```
do i = 1, n
     A(i, c1) = F(B(c2, i))
enddo
```

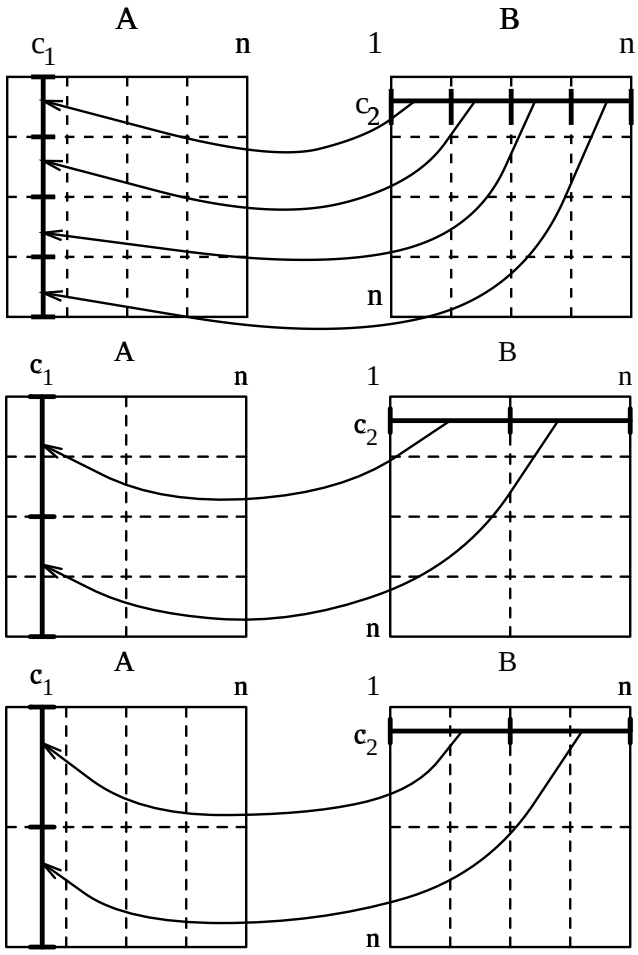


Figure 13: Variation of multiple pairs of sub-references in a loop

and B are distributed in a blocked manner on a grid with N_1 x N_2 processors. Depending on the relative values of N_1 and N_2, the best choice of communication primitive may be Scatter, Transfer, or Gather. It is not possible to determine the appropriate communication cost term just by analyzing the two pairs of sub-references individually.

The compiler has to analyze all of the sub-references varying in a loop together, to infer the relationship between simultaneous traversals of those sub-references in different processor grid dimensions. The choice of primitives is determined by the relationship between the fastest-varying `rhs` sub-reference and the fastest-varying `lhs` sub-reference. Table 3 presents some of the communication cost terms when there are two pairs of sub-references varying in a loop L. The unnumbered properties listed under the *conditions tested* column are those which must be satisfied, before an appropriate cost term is chosen, based on the numbered condition. The function `fv` chooses the faster-varying sub-reference between two given sub-references, i.e., if $\texttt{SpRatio}(s_1, s_2, L) = k$, $\texttt{fv}(s_1, s_2)$ is set to s_1 if $k \geq 1$, and otherwise to s_2. The symbol k_m appearing in various cost terms refers to the value of $\max(k_1, k_2)$. The symbols N_f and N_s refer to the number of processors in the two grid dimensions corresponding to the "faster-varying" and the "slower-varying" `rhs`

LHS	*RHS*	*Conditions Tested*	*Communication Term*
single-index (s_1) single-index (s_3)	single-index (s_2) constant (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_2, s_1 k_3-synch s_3. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 D-Scatter($\texttt{MNSP}(s_2, L)/k_m, k_m$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(s_2, L)$) D-Gather($\texttt{MNSP}(s_2, L), 1/k_m$)
single-index (s_1) single-index (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_4, s_1 k_3-synch s_3, s_2 k_4-synch s_4. 1. s_1 strictly synch s_2, s_3 strictly synch s_4. 2. $max(k_1, k_2) > 1$ 3. $max(k_1, k_2) = 1$ 4. $max(k_1, k_2) < 1$	 IDM($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$) D-Scatter($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)/k, k$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$) D-Gather($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L), 1/k$)
single-index (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_1 k_1-synch s_4, s_2 k_3-synch s_4. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 D-Scatter($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)/k_m, k_m$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$) D-Gather($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L), 1/k$)
single-index (s_1) single-index (s_3)	constant (s_2) constant (s_4)	s_1 k-synch s_3	(i) $(1 - 1/N_s) *$ Transfer(1) (ii) OneToManyMulticast(1, $\texttt{NPP}(\texttt{fv}(s_1, s_3), L)$)
single-index (s_1) constant (s_3)	constant (s_2) single-index (s_4)	s_1 k-synch s_4. 1. $k > 1$ 2. $k = 1$ 3. $k < 1$	 D-Scatter($\texttt{MNSP}(s_4, L)/k, k$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(s_4, L)$) D-Gather($\texttt{MNSP}(s_4, L), 1/k$)
constant (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_2 k-synch s_4 1. reduction op 2. default	(i) $(1 - 1/N_s) *$ Transfer(m) (ii) Reduction(1, $\texttt{NPP}(\texttt{fv}(s_2, s_4), L)$) (ii) D-Gather($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$, $\texttt{NPP}(\texttt{fv}(s_2, s_4), L)$)

Table 3: Communication terms for two pairs of sub-references varying in a Type-1 loop

sub-references respectively. The unnumbered conditions in the table ensure that all the processors in the two grid dimensions can be partitioned into mutually disjoint groups over which the indicated primitives can be carried out in parallel.

If the given sub-references do not satisfy the conditions shown in the table, the compiler first selects the `lhs` sub-reference with the higher `speed-ratio`, and similarly, the faster varying `rhs` sub-reference. Now, the `speed-ratio` is determined for those two sub-references, and the communication cost term obtained, as described earlier for the case of a single pair of sub-references varying in the loop.

Example For the example in Figure 13, the choice of communication cost term is governed by the test for k-synchronous property between sub-references $\langle A(i, c_1), 1\rangle$, and $\langle B(c_2, i), 2\rangle$, as shown in Table 3. For the three cases corresponding to 4 x 4, 4 x 2, and 2 x 4 processor grids, $\langle A(i, c_1), 1\rangle$ is determined to be 1-synchronous, 2-synchronous and 1/2-synchronous, respectively, with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop. Hence, the compiler chooses the terms Transfer($\lceil n/4 \rceil$), D-Scatter($\lceil n/4 \rceil, 2$) and D-Gather($\lceil n/4 \rceil, 2$) respectively in those cases.

4.1.3 General Cyclic distributions

All of the analysis we have described above is valid for both kinds of array distributions, block and general cyclic (cyclic and block-cyclic). For general cyclic distributions, however, another condition is added to the tests for regularity of data movement. For every sub-reference with a subscript of the form $e = \alpha_1 j_l + \beta_1$, the compiler checks if b_1 is a multiple of $s_1\alpha_1$, where s_1 is the stride and b_1 is the block size of distribution of the given dimension. The satisfaction of this condition ensures that the data elements involved in any collective communication corresponding to the given sub-reference can be accessed on the local memories of involved processors with a constant stride. Otherwise, the compiler uses cost terms corresponding to communication with a degraded performance.

4.1.4 Different/multiple loop indices

When the subscripts corresponding to a matched pair of sub-references involve different loop indices, or when one of the subscripts is of the type `multiple-index`, one simple way to analyze the data movement in terms of our earlier results would be to "freeze" (i.e., regard as Type-2) all relevant loops except for the innermost one. This would involve placing communication inside those frozen loops, and treating the corresponding loop indices as constants for the purpose of analysis. PARADIGM uses an extension of this idea, with *tiling* [29] instead of freezing of one of the outer Type-1 loops, so that communication may be combined with respect to at least the tiles of that loop.

Sub-references with different loop indices Let us first describe the analysis for a pair of sub-references of the type `single-index`, but with different values of `variation-level`. Consider sub-references sr_1 and sr_2 with subscript expressions of the form $e_1 = \alpha_1 j_q + \beta_1$, $e_2 = \alpha_2 j_p + \beta_2$, where $q > p$. If L_p is a Type-2 loop, then sr_2 is regarded as a sub-reference of the type `constant`, and the data movement is analyzed for loop L_q in a normal manner. However, if L_p is a Type-1 loop, it is assumed to be tiled suitably by the compiler. (The compiler generating communication would tile it such that the starting points of the tiles are precisely the transition points of the loop L_p for the sub-reference sr_2). In the context of this paper, we shall limit our discussion to just the part dealing with cost estimation.

The value of j_p is regarded as a constant for the purpose of analysis of communication requirements, as shown in Table 2. However, the cost term obtained is modified in the following manner to model the effect of tiling of the loop L_p. There are two cases:

1. *The sub-reference sr_2 appears on the* `rhs`: The data size of the cost term is multiplied by the tile size, given by $\texttt{MNSP}(sr_2, L_p)$, to model the effect of combining of communication with respect to the tiles of the loop L_p. Further, the term itself is multiplied by $\texttt{NPP}(sr_2, L_p)$ to account for the repetitions corresponding to the number of tiles. In fact, in this case, it is known that the cost term obtained initially is $\text{OneToManyMulticast}(1, \texttt{NPP}(sr_1, L_q))$. It is changed to $\texttt{NPP}(sr_2, L_p) * \text{OneToManyMulticast}(\texttt{MNSP}(sr_2, L_p), \texttt{NPP}(sr_1, L_q))$.
2. *The sub-reference sr_2 appears on the* `lhs`: The cost term obtained is multiplied by $\texttt{NPP}(sr_2, L_p)$ in this case too. However, the data size of the original term is left unchanged (the data corresponding to the `rhs` sub-reference is re-used in different iterations that are part of the same tile).

Some additional analysis is required when there are other pairs of sub-references that vary inside the loop L_p. Consider another pair of sub-references sr_3 and sr_4, both of the type `single-index` with a `variation-level` of m. This time, the analysis proceeds along the lines of that shown in Table 3. The compiler checks for the k-synchronous property between sr_2 and sr_3, and between sr_2 and sr_4, with respect to L_p. Tiling is used only if the k-synchronous property holds between each of the above pairs. The fastest-varying sub-reference among sr_2, sr_3, and sr_4 is selected to determine the tile size. The communication cost term is now modified using the method shown above. The following example illustrates this process.

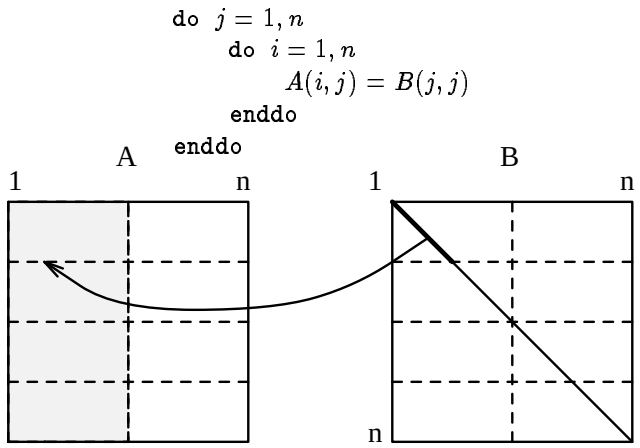


Figure 14: Example to illustrate tiling of outer loop

Example Consider the statement and the associated data movement for a 4 x 2 processor grid shown in Figure 14. The first pair of aligned dimensions have subscripts i and j varying in different loops. The j-loop is tiled, and j is regarded as a constant for the purpose of obtaining the initial term, OneToManyMulticast(1,4). There is another pair of sub-references varying in the j-loop (L_1). The sub-reference $\langle B(j,j), 1\rangle$ is 2-synchronous with both $\langle B(j,j), 2\rangle$ and $\langle A(i,j), 2\rangle$, with respect to the j-loop. Hence, the tile size is set to $\texttt{MNSP}(\langle B(j,j), 1\rangle, L_1) = \lceil n/4 \rceil$. The number of tiles is given by $\texttt{NPP}(\langle B(j,j), 1\rangle, L_1) = 4$. Therefore, the communication term is changed to $4 *$ OneToManyMulticast($\lceil n/4 \rceil, 4$).

Extension In the special case when there are two pairs of aligned sub-references, with the first pair being of the form $sr_1 = \alpha_1 j_q + \beta_1$, $sr_2 = \alpha_2 j_p + \beta_2$, and the second pair of the form $sr_3 = \alpha_3 j_p + \beta_3$, $sr_4 = \alpha_4 j_q + \beta_4$, the compiler can refine the analysis further. The data movement corresponds to a *transpose* if sr_1 is strictly synchronous with sr_4 with respect to loop L_q, and sr_2 is strictly synchronous with sr_3 with respect to loop L_p. The transpose operation is often used for changing the distribution of a two-dimensional array from row-wise distribution to column-wise distribution, or vice versa.

Sub-references with multiple loop indices For all the loops whose indices appear in a sub-reference of the type `multiple-index`, the compiler first determines those that are Type-1 loops. If there is only one such loop, the indices corresponding to the Type-2 loops are regarded as constants, and the sub-reference reduces to one of the type `single-index`. If there are two or more such loops, the compiler generating communication can again use the idea discussed above of tiling the loop surrounding the innermost of those loops to reduce communication costs. PARADIGM estimates the communication cost based on the use of tiling. Further details are given in [9].

4.1.5 Sub-references of the type `complex`

If either of the pair of sub-references varying in a Type-1 loop is of the type `complex`, the compiler is unable to infer the precise pattern of data movement in that loop. The compiler can either (i) use collective communication, which would lead to over-communication, or (ii) place communication inside the loop, and use Transfer for interprocessor data movement in each iteration, or (iii) use run-time compilation techniques, such as *inspector-executor* [24, 16], to generate communication. For the purpose of estimating communication costs, PARADIGM assumes that the compiler would use collective communication. Table 4 shows the cost term obtained for the loop when at least one of the sub-references is of the type `complex`. In the entries for

```
do i = 1, n
     A(i) = ... B(D(i)) ...
enddo
```

Figure 15: Example with a sub-reference of type `complex`

LHS	*RHS*	*Communication Term*
complex	constant	OneToManyMulticast$(1, N_I)$
constant	complex	Gather$(\lceil n/N_I \rceil, N_I)$
complex	single-index multiple-index complex	ManyToManyMulticast$(\lceil n/N_I \rceil, N_I)$
single-index multiple-index	complex	ManyToManyMulticast$(\lceil n/N_I \rceil, N_I)$

Table 4: Collective communication for sub-references involving unknowns

cost terms, n denotes the size of the corresponding array dimension, and N_I denotes the number of processors in the grid dimension. In this case, there is no special analysis needed for multiple pairs of sub-references varying in a loop. Those pairs are analyzed independently, and the resulting terms are composed in the normal manner.

Example In the program segment shown in Figure 15, if A and B are distributed on N processors, the communication cost for B is estimated as ManyToManyMulticast$(n/N, N)$. This corresponds to placing communication outside the loop without any run-time processing and using collective communication. If the compiler chooses not to move communication outside the loop, the cost would be different, as discussed in the next section. In the above example, the compiler adds another cost term for broadcasting D to all processors, if D is not already replicated on all processors.

4.1.6 Extensions: Replication and Mapping to a Constant Processor Position

We now briefly describe how our analysis extends to the case when at least one of the distributed array dimensions is not partitioned in a block or general cyclic manner. Let sr_1 and sr_2 be the `lhs` and the `rhs` sub-references respectively, with the `variation-level` corresponding to Type-1 loop L.

- When the `rhs` dimension is replicated, all data movement is internal to that dimension. The cost term contributed is IDM(`MNSP`(sr_2, L)).
- When the `lhs` dimension is replicated, the analysis proceeds as shown above in Table 4 for the case when `lhs` sub-reference is of the type `complex`.
- When the `lhs` or the `rhs` dimension is mapped to a constant processor position, the selection of communication primitives is done exactly as for the case when the corresponding subscripts are of the type `constant`. If the `rhs` dimension is mapped to a constant processor position, the data size of the cost term is given by `MNSP`(sr_2, L).

In order to simplify the description of our analysis, we will not discuss these distributions any further.

4.2 Cost Terms for Single Data Movement

A pair of sub-references of the type `constant` or varying in a Type-2 loop, represents either an internalized data movement, or communication corresponding to a single Transfer. Hence, the analysis for obtaining a communication cost term consists of determining (i) if any interprocessor communication takes place, and (ii) if so, how often the communication takes place. If the two sub-references are of the type `constant` or `single-index`, the compiler checks for the strictly-synchronous property between them. The presence of strictly synchronous property implies that all data movement is internal, and no communication cost term is needed. If any sub-reference is of the type `multiple-index` or `complex`, the compiler infers the absence of communication only if both the subscripts are identical, and both the array dimensions have identical distributions. If communication is needed, the Transfer primitive will actually be invoked only for those instances (corresponding to the iterations of Type-2 loops) when the data movement across the grid dimension is not internal to a processor. PARADIGM considers two cases:

Case 1: Boundary Communication Both sub-references, sr_1 and sr_2 are of the type single-index, and satisfy the boundary-communication test, discussed in Section 2. The number of iterations of the corresponding Type-2 loop, L, during which an interprocessor Transfer takes place is given by $\texttt{BD}(sr_1, sr_2, L) * (\texttt{NPP}(sr_2, L) - 1)$. The term $\texttt{NPP}(sr_2, L) - 1$ denotes the number of processor boundaries across which communication takes place, while $\texttt{BD}(sr_1, sr_2, L)$ denotes the number of data items that are transferred (in different iterations) across each processor boundary. The expressions for these terms have been shown earlier in Equations 1 and 4 respectively. Since communication is placed *inside* the loop L, and the obtained cost term would be multiplied by the number of iterations of that loop, say, $n_{l'}$, the normalized cost term is given by $(\texttt{BD}(sr_1, sr_2, L) * (\texttt{NPP}(sr_2, L) - 1) * \text{Transfer}(1))/n_{l'}$.

Case 2: General Communication This is the default case in which cost estimate is based on the following simplifying assumption. Any two arbitrary elements along the given array dimensions distributed over N_k processors are assumed to belong to the same processor with a probability of $1/N_k$. Hence, the probability that a Transfer is needed during an arbitrary loop iteration is $1 - 1/N_k$, and the cost term is obtained as $(1 - 1/N_k) * \text{Transfer}(1)$. This elegantly takes care of the important boundary case $N_k = 1$, since the communication cost term does evaluate to zero when the array dimensions are collapsed.

Example In Figure 15, if the compiler were to place communication for B inside the loop (to avoid over-communication or because of lack of space for buffer holding non-local data), the cost for a *single* loop iteration would be estimated as $(1 - 1/N) * \text{Transfer}(1)$.

4.3 Composition of Communication Terms

Once the communication terms corresponding to all the grid dimensions have been obtained, they are composed together to obtain the overall estimate of communication cost. This process consists of the following steps: (i) combining terms with the same communication primitive, (ii) determining the order in which different primitives are invoked, and obtaining the data sizes of the terms, and (iii) multiplying the terms by the expected number of repetitions of that communication during the program.

4.3.1 Communication Terms with Same Primitive

The terms involving the same primitive in different grid dimensions are combined as shown in Table 5. The combined term represents the primitive carried out over a bigger group, spanning multiple grid dimensions. As discussed earlier, a Shift term always appears in conjunction with an IDM term. Figure 16 illustrates the composition of two `shifts` in different dimensions together with the associated terms for internalized data

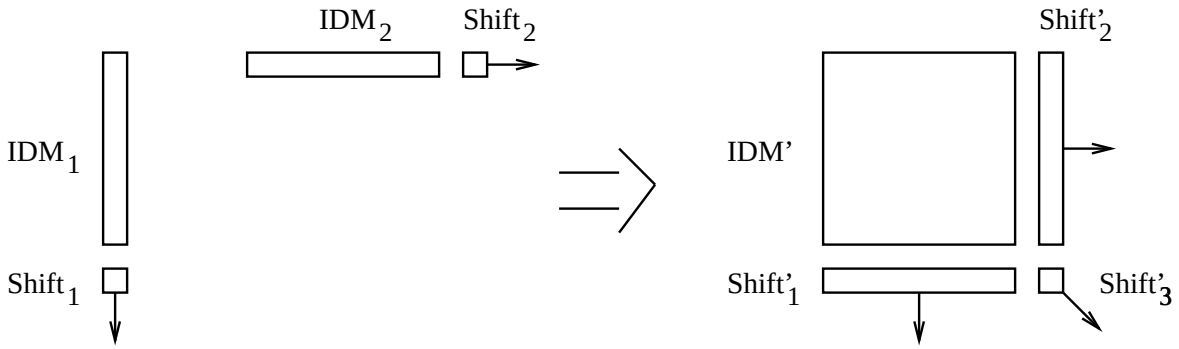


Figure 16: Composition of Shift and IDM pairs

Term1	*Term2*	*Resultant Term*
IDM(m_1)	IDM(m_2)	IDM($m_1 * m_2$)
Gather(m_1, p_1)	Gather(m_2, p_2)	Gather($m_1 * m_2, p_1 * p_2$)
IDM(m_1), ($N_1 > 1$) Shift(c_1)	IDM(m_2), ($N_2 > 1$) Shift(c_2)	IDM($m_1 * m_2$), ($N_1 > 1$) Shift($c_1 * m_2$), ($N_2 > 1$) Shift($c_2 * m_1$), ($N_1 > 1$ & $N_2 > 1$) Shift($c_1 * c_2$)
Reduction(m_1, p_1)	Reduction(m_2, p_2)	Reduction($m_1 * m_2, p_1 * p_2$)
ManyToManyM'cast(m_1, p_1)	ManyToManyM'cast(m_2, p_2)	ManyToManyM'cast($m_1 * m_2, p_1 * p_2$)
OneToManyM'cast(m_1, p_1)	OneToManyM'cast(m_2, p_2)	OneToManyM'cast($m_1 * m_2, p_1 * p_2$)
$(1 - 1/p_1) *$ Transfer(m_1)	$(1 - 1/p_2) *$ Transfer(m_2)	$(1 - 1/(p_1 * p_2)) *$ Transfer($m_1 * m_2$)

Table 5: Combining of terms with identical communication primitives

movement. The composition leads to `shifts` in the original directions with bigger data sizes, a "diagonal" `shift`, and a purely internalized data movement.

4.3.2 Ordering of Primitives

Since the primitives corresponding to different terms implement the data movement in distinct grid dimensions, they can legally be composed in any order. However, the order in which they are invoked is important because the position of each primitive affects the message sizes and the number of processors involved (in parallel) in subsequent primitives. It is desirable to obtain an ordering that leads to fewer processors being involved and smaller messages handled by each processor, but sometimes, there is a trade-off between the two. For example, consider the statement shown in Figure 17, where the arrays A and B are distributed in an identical manner on a 2-D grid. The primitives required are: Gather in the first dimension, and Transfer in the second dimension. Figure 18 illustrates the two possible orderings for a 3 x 3 grid. If Gather is invoked first, it is carried out with a data size of $n/3$ words, over 3 processors, followed by a single Transfer of n words of data. If this ordering is reversed, there are 3 parallel Transfers that take place, each involving $n/3$ words, followed by a Gather operation, also with a data size of $n/3$ words, over 3 processors. The second ordering in the above example leads to the use of parallelism in implementing communication, and would

```
do i = 1, n
     A(n, n) = F(B(i, 1))
enddo
```

Figure 17: Statement requiring Gather and Transfer in different dimensions

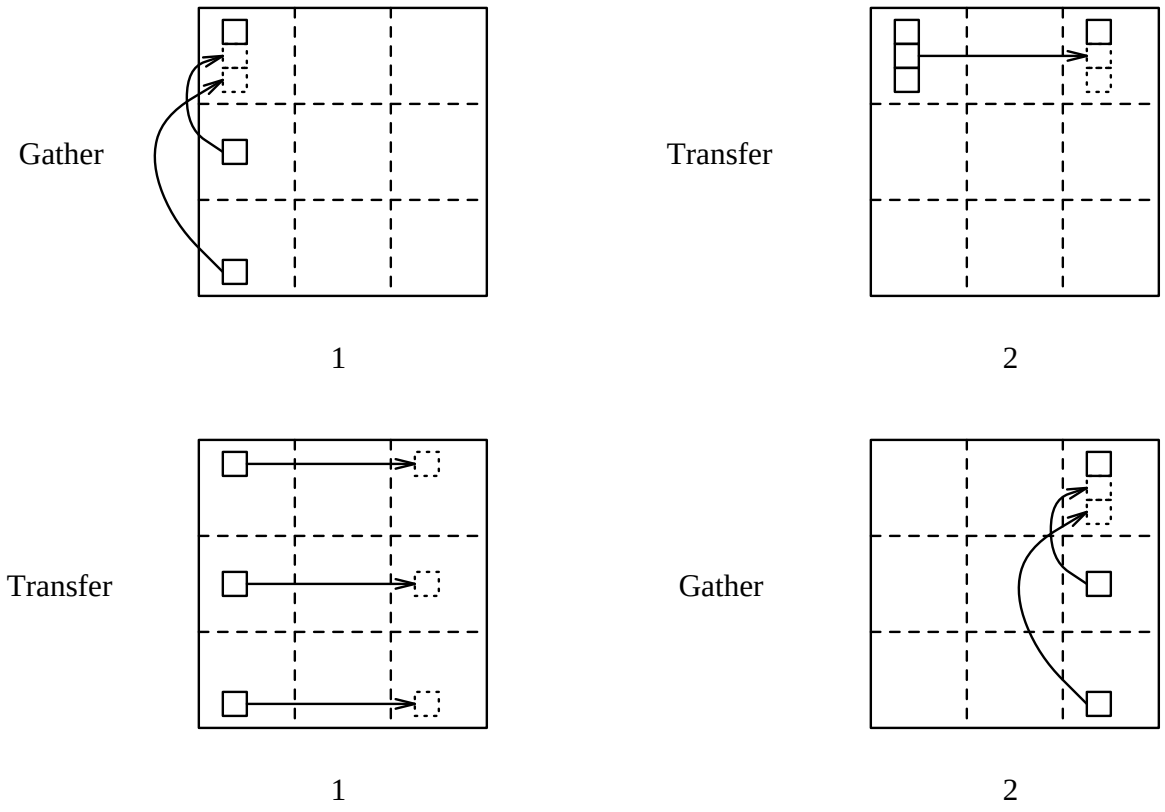


Figure 18: Possible compositions of Gather and Transfer

yield better performance if there were no other communications being carried out on the grid of processors. This suggests resolving the trade-off in favor of reducing the message sizes handled by processors. When there is no trade-off involved, the compiler should use an ordering that reduces the message sizes and/or the number of processors involved. These considerations suggest the ordering shown in Table 6.

Determination of message sizes

The data size associated with each communication term initially represents just one "edge" of the overall volume of data being communicated. In accordance with the ordering of primitives shown above, the compiler determines the actual message size for each primitive corresponding to a communication term. First, the compiler determines the product of individual data sizes associated with each communication term, including an IDM term. This product represents the complete volume of data to be communicated, and serves as the

Rank	*Primitive*	*Message Size*	*No. of Processors*
1	Reduction	reduced	reduced
2	Scatter	reduced	increased
3	Shift, Transfer	preserved	preserved
4	OneToManyMulticast	preserved	increased
5	Gather	increased	reduced
6	ManyToManyMulticast	increased	increased

Table 6: Ordering of communication primitives for composition

```
do j = 1, n
    do i = 1, n
        A(i, j, c1) = F(B(i, c2, c3))
    enddo
enddo
```

Figure 19: Example of composition of communication terms

message size for each term corresponding to any of the first five primitives listed in Table 6. (Note that the data sizes used in terms for the Reduction and Scatter primitives already reflect the "reduction" in message size caused by those primitives, as can be seen from the entries shown in Tables 2 and 3). If there are terms present for both Gather and ManyToManyMulticast primitives, the data size for the ManyToManyMulticast is multiplied further by the number of processors shown in the Gather term, to account for the increased amount of data participating in that primitive following the Gather operation.

Example Consider the statement shown in Figure 19, where the arrays, A and B are distributed in an identical manner on a N_1 x N_2 x N_3 grid. The communication terms obtained are: IDM($\lceil n/N_1 \rceil$), OneToManyMulticast($1, N_2$), and $(1 - 1/N_3) * \text{Transfer}(1)$. These terms are composed together to give the communication cost estimate as $(1 - 1/N_3) * \text{Transfer}(\lceil n/N_1 \rceil) + \text{OneToManyMulticast}(\lceil n/N_1 \rceil, N_2)$.

4.4 Number of Repetitions

The sequence of communication primitives, placed outside loop L_{l+1} as shown in Figure 5, is repeated during iterations of the surrounding Type-2 loops, L_1 through L_l. Due to the presence of conditionals in the program, the flow of control may not reach that point during every iteration. The number of times the flow of control reaches the statement corresponding to loop L_{k+1} can be expressed as:

$$num(L_{k+1}) = \rho(L_{k+1}) * (\prod_{k=1}^{l} n_k),$$

where $\rho(L_{k+1})$ is the *reaching probability* [9] of that statement, and n_k is the iteration count of the loop L_k. The compiler multiplies the cost of the sequence of communication primitives obtained above by $num(L_{k+1})$, to reflect the overall cost of that communication for the entire program. The value of $num(L_{k+1})$ may not be known at compile time due to the presence of conditionals and symbolic loop bounds. A good method for obtaining estimates of the values of $num(L_{k+1})$ is to use profiling [25, 6]. As mentioned earlier, PARADIGM relies on input from the user to obtain the values of any unknown loop bounds and reaching probabilities of statements. Unless explicitly supplied a value, PARADIGM assumes a reaching probability of one for each statement.

The above analysis assumes strict sequentiality in the process of carrying out communication inside the Type-2 loops. In practice, the generation of messages may sometimes get pipelined over different iterations of the outer loops. In some special cases, the compiler can detect this pipelining and refine the cost estimate [9]. Those details are beyond the scope of this paper.

5 Results

We now present results on automatic data partitioning that show the importance of using good performance estimates to guide those decisions. These results are obtained by PARADIGM using the analysis described

in this paper for estimating communication costs. While the analysis we have presented is largely machine-independent, the actual values of machine-specific parameters have been supplied for an Intel iPSC/2 hypercube with 16 processors. We will show performance results on that machine to evaluate the quality of data partitioning decisions made by the compiler.

A discussion of the techniques used by PARADIGM to avoid searching exhaustively through the space of data partitioning schemes while using these estimates is beyond the scope of this paper. Those details and more extensive results on automatic data partitioning are presented in [9, 11]. In this section, we shall discuss results from two representative programs that show a range of data partitioning decisions which require more effective techniques for estimating communication costs than those currently used by most other compilers. The first program, `Tred2` is a routine from the EISPACK library, it reduces a real symmetric matrix to a symmetric tridiagonal matrix. `Jacobi` is a simplified version of a relaxation code that performs Jacobi iterations in a loop.

5.1 Application 1: TRED2

The source code of `tred2` is listed in Figure 20, where statements are labeled by the line numbers. We have used a data size of $N = 512$ for the program supplied to the compiler.

Alignment One of the most interesting decisions for the program relates to the alignment of array dimensions, as different parts of the program give rise to conflicting requirements on alignment. For example, the references in statements 58, 71 and 77 favor alignment of D_1 with Z_1, while those in statements 18, 59 and 83 favor alignment of D_1 with Z_2. Statement 4 favors alignment of D_1 with A_2, for which there is a further preference for alignment with Z_2. PARADIGM represents these alignment constraints using a *component affinity graph* [18], in which the nodes represent array dimensions, and weighted edges represent alignment preferences. If the edge weights were obtained simply as a count of the number of statements favoring that alignment, an optimal decision would be to mutually align A_2, Z_2, D_1 and Z_1, and to align A_1 with Z_1. However, using the techniques described in this paper, PARADIGM obtains edge weights as estimates of the extra communication cost due to mis-alignment of array dimensions. That leads to a decision to align A_1, Z_1, D_1 and E_1 together, and to align A_2 with Z_2 [9].

Block/Cyclic Distribution There are a number of statements in the program that favor cyclic distribution of all array dimensions, and a sole statement favoring block distribution. The reference on statement 16 requires nearest-neighbor communication, and hence there is a preference recorded for block distribution of D_1 and E_1, to cut down on the volume of communication [11]. However, our estimation process shows that the extra communication cost due to cyclic distribution is much smaller than the expected performance gains due to better load balance [9]. Therefore, the compiler chooses cyclic distribution for all array dimensions.

Number of Processors At a number of statements, the given program exhibits parallelism at two loop levels that traverse the rows and the columns of arrays A and Z. The program also shows communication across both the dimensions of arrays, if distributed. Hence, it is difficult to determine the relative number of processors on which to distribute the array dimensions, without using communication and computational cost estimates. Based on these estimates, PARADIGM compares different N_1 x N_2 grid configurations, varying N_1 from 1 to 16, doubling its value in each step. It selects 16 x 1 as the best grid configuration. These decisions lead to the distribution of arrays A and Z by *rows* in a *cyclic* manner, and of arrays D and E also in a cyclic manner, on all the 16 processors.

Evaluation In order to evaluate each data partitioning decision described above, we implemented different parallel versions of `tred2` on the iPSC/2. Each version was obtained by hand-simulating the compilation process under a different data partitioning scheme for that program. The execution times observed on the

```
1     DO 5 I = 1, N
2           DO 3 J = I, N
3             Z(J,I) = A(J,I)
4           D(I) = A(N,I)
5     CONTINUE
6     IF (N .EQ. 1) GO TO 82
7     DO 63 II = 2, N
8           I = N + 2 - II
9           L = I - 1
10          H = 0.0D0
11          SCALE = 0.0D0
12          IF (L .LT. 2) GO TO 16
13          DO 14 K = 1, L
14            SCALE = SCALE + DABS(D(K))
15          IF (SCALE .NE. 0.0D0) GO TO 23
16          E(I) = D(L)
17          DO 21 J = 1, L
18                D(J) = Z(L,J)
19                Z(I,J) = 0.0D0
20                Z(J,I) = 0.0D0
21          CONTINUE
22          GO TO 62
23          DO 25 K = 1, L
24                D(K) = D(K) / SCALE
25                H = H + D(K) * D(K)
26          CONTINUE
27          F = D(L)
28          G = -DSIGN(DSQRT(H),F)
29          E(I) = SCALE * G
30          H = H - F * G
31          D(L) = F - G
32          DO 33 J = 1, L
33            E(J) = 0.0D0
34          DO 45 J = 1, L
35                F = D(J)
36                Z(J,I) = F
37                G = E(J) + Z(J,J) * F
38                JP1 = J + 1
39                IF (L .LT. JP1) GO TO 43
40                DO 43 K = JP1, L
41                      G = G + Z(K,J) * D(K)
42                      E(K) = E(K) + Z(K,J) * F
43                CONTINUE
44                E(J) = G
45          CONTINUE
46          F = 0.0D0
47          DO 50 J = 1, L
48                E(J) = E(J) / H
49                F = F + E(J) * D(J)
50          CONTINUE
51          HH = F / (H + H)
52          DO 53 J = 1, L
53            E(J) = E(J) - HH * D(J)
54          DO 61 J = 1, L
55                F = D(J)
56                G = E(J)
57                DO 58 K = J, L
58                      Z(K,J) = Z(K,J) - F * E(K) - G * D(K)
59                D(J) = Z(L,J)
60                Z(I,J) = 0.0D0
61          CONTINUE
62          D(I) = H
63    CONTINUE
64    DO 81 I = 2, N
65          L = I - 1
66          Z(N,L) = Z(L,L)
67          Z(L,L) = 1.0D0
68          H = D(I)
69          IF (H .EQ. 0.0D0) GO TO 78
70          DO 71 K = 1, L
71            D(K) = Z(K,I) / H
72          DO 78 J = 1, L
73                G = 0.0D0
74                DO 75 K = 1, L
75                  G = G + Z(K,I) * Z(K,J)
76                DO 78 K = 1, L
77                  Z(K,J) = Z(K,J) - G * D(K)
78          CONTINUE
79          DO 80 K = 1, L
80            Z(K,I) = 0.0D0
81    CONTINUE
82    DO 85 I = 1, N
83          D(I) = Z(N,I)
84          Z(N,I) = 0.0D0
85    CONTINUE
86    Z(N,N) = 1.0D0
87    E(1) = 0.0D0
88    END
```

Figure 20: Source code of `tred2` routine

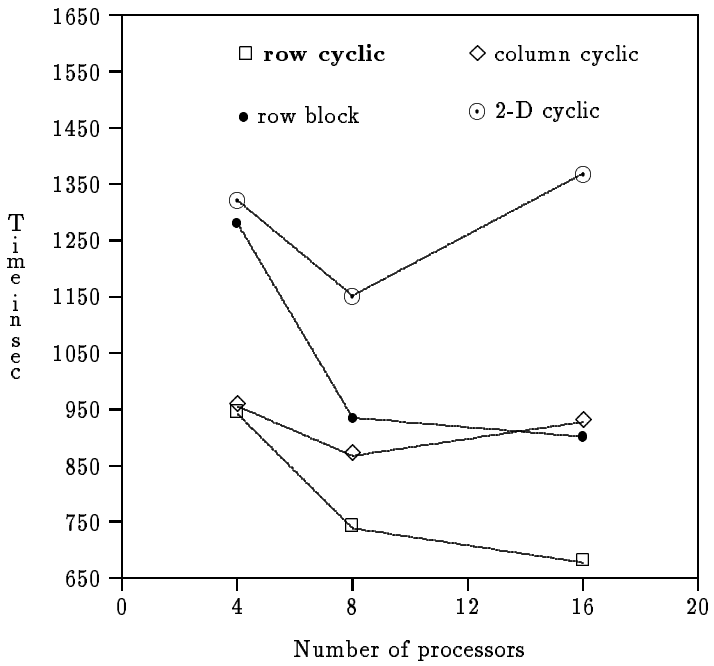


Figure 21: Performance of `tred2` on Intel iPSC/2 for data size $n = 512$

iPSC/2 are shown in Figure 21. The sequential time for the program is not shown because the program could not be run on a single node due to memory limitations. The data distribution scheme selected by PARADIGM, marked *row cyclic*, performs the best. A different alignment, i.e., of D_1 and E_1 with A_2 and Z_2, leads to the distribution of columns of A and Z (and arrays D and E) in a cyclic manner. This scheme, marked *column cyclic*, does not perform as well, confirming that the alignment decision made by PARADIGM, based on communication cost estimates, is a better one. The other two schemes, referred to as *row block* and *2-d cyclic* correspond to departures from the selected data partitioning regarding decisions of cyclic distribution, and of collapsing one of the dimensions. The poorer performance resulting from these distributions confirms these other decisions as well.

5.2 Application 2: JACOBI

The source code of the `Jacobi` program is shown in Figure 22. This simplified version repeatedly carries out relaxation computations followed by copying of array elements to store the elements computed in the previous step.

The decisions to align A_1 with B_1, and A_2 with B_2, and to distribute each array dimension in a blocked manner are straightforward, with no conflicting requirements [9]. However, the compiler needs performance estimates to determine the relative number of processors on which to distribute the two array dimensions. A trivial solution to this problem cannot be obtained because there is parallelism to be exploited and communication in both the grid dimensions. The contribution of communication cost to the program execution time is estimated by the compiler as:

$$\text{Communication cost} \quad = \quad ncycles * (2 * (N_1 > 1) * \text{Shift}(n/N_2) + 2 * (N_2 > 1) * \text{Shift}(n/N_1))$$

Let us consider two grid configurations, 1 x 16, and 4 x 4. The first of these corresponds to a column partitioning of the arrays. It leads to two Shift operations being carried out in every cycle, with a data size of n words each. The second configurations corresponds to a "2-D" partitioning of the arrays, where both the rows and the columns are distributed on 4 processors each. That leads to four Shift operations in

```
parameter (np2 = 514, ncycles = 100)
double precision A(np2,np2), B(np2,np2)

np1 = np2 - 1
do k = 1, ncycles
    do i = 2, np1
        do j = 2, np1
            B(i,j) = 0.5 * A(i,j) + 0.125 * (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
        enddo
    enddo
    do i = 2, np1
        do j = 2, np1
            A(i,j) = B(i,j)
        enddo
    enddo
enddo
```

Figure 22: Source code for `Jacobi` program

Data Size n	*Column Blocked Time (s)*	*2-D Blocked Time (s)*	*Column Cyclic Time (s)*
64	1.28	1.45	1.66
128	4.10	4.12	5.82
256	15.39	14.87	23.29
512	64.43	59.66	96.64
1024	257.84	243.24	386.94

Table 7: Performance of different versions of Jacobi on iPSC/2

every cycle, each with a data size of $n/4$ words. For smaller values of n, the first scheme is better, since the start-up costs of sending messages dominate the communication cost. As the data size is increased, the second scheme starts becoming more attractive, since the total amount of data transferred is lower under that scheme. Given different values of n that are doubled in each step, PARADIGM starts choosing the second scheme at $n = 1024$.

The execution times obtained for each of those versions running on the 16-processor iPSC/2 are shown in Table 7. These results confirm that the 2-D partitioning starts performing better than the column partitioning for larger array sizes. Similar results have been reported earlier by other researchers [3]. We have also presented results for cyclic distribution of columns to show the performance penalty if the compiler did not choose block distribution.

While these results confirm the compiler's prediction regarding the suitability of the 2-D partitioning at larger data sizes, the actual data size at which that scheme starts performing better than column partitioning is not predicted very accurately. We believe that this difference between the predicted value ($n = 1024$) and the observed value ($n = 256$) is primarily due to the cost function used for the Shift operation being inaccurate. The primary focus of our work in estimating communication costs has been to obtain the estimates in terms of cost functions of various communication primitives (which is performed satisfactorily in this case). Given more accurate performance characteristics of such primitives, obtained by approaches proposed in the literature, such as the "training set" method [3], we believe that PARADIGM could do an even better job of selecting good data partitioning schemes.

6 Conclusions

We have presented a methodology for estimating communication costs in a program, to guide data partitioning decisions. A significant feature of our approach is that the analysis is performed on the original source program in global address space. This allows various alternatives regarding data partitioning to be evaluated without actually having to generate code for each version. At the same time, the compiler accounts for important communication optimizations to maintain accuracy of the analysis.

Another contribution of this work has been to develop a systematic analysis for characterizing data movement. The notion of synchronous properties and speed-ratio between sub-references varying in a loop helps identify the nature of data movement, and hence helps select a collective communication primitive. The other traversal properties of each individual sub-reference help estimate the size of messages and the number of processors involved in communication. We have also described various issues and trade-offs in composing communication across multiple processor grid dimensions, when arrays have more than one distributed dimension. On the one hand, this analysis offers the advantage of more precise information about the global state of communication, which is useful for estimating performance, and on the other hand, it enables enhanced performance through the use of collective communication primitives [10] for a compiler generating communication [12].

The methodology presented in this paper has been implemented as part of the PARADIGM compiler. We have shown that detailed communication cost estimates, as obtained by our methodology, are essential for good data partitioning decisions in a number of cases. Our performance results so far indicate that these estimates do guide data partitioning decisions in the right direction.

There are many limitations of our present work, and a number of ways in which it can be extended. Our approach in dealing with program parameters that are unknown at compile time is simplistic, and can benefit by incorporating ideas from the work of Fahringer and Zima [5, 6]. Our approach can also be extended to consider communication costs of multiple references at the same time, and to incorporate additional compiler optimizations like coalescing messages [13], and recognizing pipeline communication, as done by PTRAN-II [12].

Acknowledgements

We are extremely grateful to the anonymous referees for their valuable comments.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Williamsburg, VA, April 1991.

[4] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proc. Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[5] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, September 1993.

[6] T. Fahringer and H. P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[7] P. Feautrier. Towards automatic data distribution. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[8] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[9] M. Gupta. *Automatic data partitioning on distributed memory multicomputers.* PhD thesis, University of Illinois, September 1992.

[10] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[11] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[12] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K.Y. Wang, and M. Burke. PTRAN II: A compiler for High Performance Fortran. In *Proc. 4th Workshop on Compilers for Parallel Computers*, Delft, Netherlands, December 1993.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[14] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Rice University, April 1991.

[15] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[16] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[17] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proc. Supercomputing '90*, New York, NY, November 1990.

[18] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[19] Message Passing Interface Forum. *Document for a standard message-passing interface*, November 1993. (Draft), available from University of Tennessee, Knoxville, Tennessee.

[20] M. O'Boyle and G. A. Hedayat. A transformational approach to compiling Sisal for distributed memory architectures. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[21] M.J. Quinn and P. J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7:69–76, September 1990.

[22] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–481, October 1991.

[23] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[24] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[25] V. Sarkar. Determining average program execution times and their variance. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[26] E. Su, D. J. Palermo, and P. Banerjee. Automating parallelization of regular computations for distributed memory multicomputers in the PARADIGM compiler. In *Proc. 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[27] K.-Y. Wang. *Intelligent program optimization and paralleilzation for parallel computers.* PhD thesis, Purdue University, April 1991.

[28] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[29] M. J. Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, Reno, Nevada, November 1989.

[30] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[31] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

A Proof of Correctness of Theorem 1

In order to prove our results regarding the synchronous properties, we need the following two lemmas.

Lemma 1 *Given any integer x, and positive integers y and z,*

$$\lfloor x/(y*z) \rfloor = \lfloor \lfloor x/y \rfloor / z \rfloor$$

Proof

Let	x	$=$	$q*y+r$,	where $0 \le r < y$, q and r are integers.
Let	q	$=$	$q'*z+r'$,	where $0 \le r' < z$, q', and r' are integers.
We have	$\lfloor x/(y*z) \rfloor$	$=$	$\lfloor (q*y+r)/(y*z) \rfloor$	
		$=$	$\lfloor q/z + r/(y*z) \rfloor$	
		$=$	$\lfloor (q'*z+r')/z + r/(y*z) \rfloor$	
		$=$	$\lfloor q' + r'/z + r/(y*z) \rfloor$	
		$=$	$q' + \lfloor (r'*y+r)/(y*z) \rfloor$	
Now	$r'*y+r$	$<$	$(r'+1)*y$	(since $r < y$)
	$r'+1$	$\le$	z	(since $r' < z$)
Hence,	$r'*y+r$	$<$	$z*y$	
	$\lfloor (r'*y+r)/(y*z) \rfloor$	$=$	0	
Therefore,	$\lfloor x/(y*z) \rfloor$	$=$	q'	
		$=$	$\lfloor q/z \rfloor$	
		$=$	$\lfloor \lfloor x/y \rfloor / z \rfloor$	

□

Lemma 2 *Given any integer x, and positive integers y and z,*

$$\lfloor x_1/y_1 \rfloor = \lfloor x_2/y_2 \rfloor \quad \Rightarrow \quad \lfloor x_1/(z*y_1) \rfloor = \lfloor x_2/(z*y_2) \rfloor$$

Proof

$$\begin{array}{lllll} & \lfloor x_1/y_1 \rfloor & = & \lfloor x_2/y_2 \rfloor & \\ \Rightarrow & \lfloor \lfloor x_1/y_1 \rfloor /z \rfloor & = & \lfloor \lfloor x_2/y_2 \rfloor /z \rfloor & \\ \Rightarrow & \lfloor x_1/(z*y_1) \rfloor & = & \lfloor x_2/(z*y_2) \rfloor & \text{(by Lemma 1)} \end{array}$$

□

Proof of Theorem 1 During loop iteration i, the two sub-references are mapped to positions $\lfloor (s_1(\alpha_1 i + \beta_1) - o_1)/b_1 \rfloor [\mathrm{mod} N_k]$ and $\lfloor (s_2(\alpha_2 i + \beta_2) - o_2)/b_2 \rfloor [\mathrm{mod} N_k]$ respectively. To prove the strictly synchronous property, it is sufficient to show that

$$\lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/b_1 \rfloor = \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/b_2 \rfloor \qquad \forall i \in I$$

The first set of conditions implies that

$$\begin{array}{lllll} & (s_1\alpha_1 i)/b_1 + (s_1\beta_1 - o_1)/b_1 & = & (s_2\alpha_2 i)/b_2 + (s_2\beta_2 - o_2)/b_2, & \forall i \in I \\ \Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/b_1 \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/b_2 \rfloor & \text{(Q.E.D. for case 1)} \end{array}$$

The condition (iii) from the second set implies that

$$\begin{array}{lllll} & \lfloor (s_1\beta_1 - o_1)/s_1\alpha_1 \rfloor + i & = & \lfloor (s_2\beta_2 - o_2)/s_2\alpha_2 \rfloor + i, & \forall i \in I \\ \Rightarrow & \lfloor (s_1\beta_1 - o_1)/s_1\alpha_1 + i \rfloor & = & \lfloor (s_2\beta_2 - o_2)/s_2\alpha_2 + i \rfloor, & \\ \Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/s_1\alpha_1 \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/s_2\alpha_2 \rfloor & \\ \Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/(m * s_1\alpha_1) \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/(m * s_2\alpha_2) \rfloor & \text{(by Lemma 2)} \\ \Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/b_1 \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/b_2 \rfloor & \text{(from conditions (i) and (ii))} \end{array}$$

□

Performance Prediction and Scalability Analysis for Data Parallel Programs

Celso L. Mendes, Jhy-Chun Wang and Daniel A. Reed
University of Illinois at Urbana-Champaign

With the advance of parallel processing and data parallel languages (e.g., HPF), an application software developer's mental programming model becomes distantly different from the actual code that executes on a particular parallel system. Constructing a high-performance parallel program requires a cycle of time-consuming experimentation and refinement. Given the programming difficulty and cost, it is critical to understand the potential scalability of a code as a function of number of processors (P) and problem sizes (N). It is equally important that the understanding of program behavior can be traced back to the high-level data parallel source code, not just to the SPMD code executed on the parallel system.

In this paper, we propose a symbolic expression oriented scalability analysis model. Under this model, we express the scalability of code fragments in a data parallel program as analytical functions of N and P. By executing an instrumented version of the program, we derive the coefficients for the model, taking instrumentation overhead into account. Predicting the performance of the noninstrumented program for different numbers of processors or problem sizes then becomes a simple evaluation of the corresponding functions for the given values of N and P. Our point is to show that instrumentation, combined with compiler support, provides the necessary framework to successfully perform quick predictions under scaling of the problem or of the system.

This model is part of an integration system of Rice Fortran 77D compiler and Illinois Pablo Environment. That integrated system permits analysis and correlation of runtime performance with the original data parallel source code. Thus, our goal is to provide a scalability analysis tool in terms of high-level source code, offering an easier understanding of program behavior for general application developers. We present preliminary results and the current status of our work.

Automatic Performance Prediction and Scalability Analysis for Data Parallel Programs

Celso L. Mendes* Jhy-Chun Wang† Daniel A. Reed‡

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Email: {mendes,jcwang,reed}@cs.uiuc.edu

Abstract

With data parallel languages like High Performance Fortran (HPF), an application developer's mental model of program execution can differ markedly from the code that actually executes on a particular parallel system. In such situations, understanding the reasons for poor program performance requires a time consuming cycle of performance experimentation and code tuning where one seeks to understand the relation between data parallel source and the performance of compiler-synthesized SPMD code. Because performance is the primary motivation for parallel computing, it is critical to understand the potential scalability of a code as a function of the number of processors (P) and problem size (N). It is equally important that this understanding be related to the high-level data parallel source code, not just to the SPMD code.

In this paper, we propose a performance scalability model for data parallel code fragments based on symbolic expressions of N and P. In this approach, the data parallel compiler synthesizes instrumented SPMD code and records sufficient compile-time data to relate the dynamic performance of the SPMD code to the original data parallel source. By correlating this data, we can derive coefficients for the original data parallel program's scalability model. We present preliminary results of our model with two examples written in Fortran D, a precursor to HPF.

1 Introduction

Despite the performance potential of distributed memory parallel systems, several factors have limited their widespread adoption. Of these, high performance variability and the large programming effort required to write explicitly parallel message passing code are among the most significant. Data parallel languages, like HPF [8] and its precursor, Fortran D [7], have been proposed as mechanisms to lessen this parallel programming burden. By allowing the programmer to construct a parallel application at a semantic higher level, without recourse to low-level message passing code, HPF is an effective specification language for regular, data parallel algorithms.

Moreover, after investing substantial intellectual effort in developing a parallel program, its execution may yield only a small fraction of peak system performance. And, even if the data parallel code is portable across multiple parallel architectures, it is highly unlikely that it will achieve high performance on all architectures. Finally, even on a single parallel architecture, observed application performance may vary substantially as a function of input parameters.

*Supported by CNPq/Brazil, process 280005/94-6(NV).

†Supported in part by the Advanced Research Projects Agency under ARPA contract number DAVT63-91-C-0029.

‡Supported in part by the Advanced Research Projects Agency under ARPA contract number DAVT63-91-C-0029 and by the National Science Foundation under grant NSF IRI 92-12976.

To understand the causes for performance variability in data parallel codes, one needs high-level performance analysis tools and techniques. Unfortunately, most current performance tools are targeted at the collection and presentation of program performance data when the parallelism and interprocessor communication are explicit and the program execution model closely mimics that in the source code (i.e., as is the case for message passing codes). For data parallel languages like HPF, such tools can only capture and present dynamic performance data in terms of primitive operations (e.g., communication library calls) in the compiler-generated code; clearly, this falls far short of the ideal. At a minimum, to support source-level performance analysis of programs in data parallel languages, compilers and performance tools must cooperate to integrate information about the program's dynamic behavior with compiler knowledge of the mapping from the low-level, explicitly parallel code to the high-level source.

More generally, mechanisms for predicting data parallel program performance as a function of application and architecture parameters would further lessen the high intellectual cost of developing parallel applications by allowing application developers to understand the performance implications of data parallel code constructs. Important questions include determining how application performance changes with variations in the parallel system configuration or application problem size, and identifying which code fragments will become the performance limiting bottlenecks as hardware or application parameters change. These predictions need not be quantitatively exact, only qualitatively exact (i.e., it is acceptable for the magnitude of prediction errors to be as high as 10–20 percent if the predictions identify bottlenecks and track their movements).

In this paper, we propose a technique for predicting the performance scalability of data parallel programs as a function of the number of processors and of the problem size. This scalability prediction is based on compiler-derived program transformation data and on dynamic performance data captured during execution of the compiler-synthesized SPMD code. From these two sources we derive a scalability model that consists of symbolic expressions representing the execution complexity of individual program sections. Predicting performance for a program and architecture configuration is then a simple evaluation of the symbolic expressions.

Although we use an instrumented version of the program to determine the performance model parameters, our technique compensates for performance perturbations due to instrumentation by considering the effects of software instrumentation costs. Thus, we predict the performance of non-instrumented codes using performance data from the instrumented versions.

Our current model is based on an integrated compilation and performance analysis system jointly developed by the Rice Fortran D group and the University of Illinois Pablo group [1]. Compile-time data on program transformations and the relation of generated SPMD code to high-level data parallel source allows us to link dynamic performance data with the corresponding high-level program. Using this linkage, we then build parametrized execution models for code fragments in the original Fortran D source code.

The remainder of this paper is organized as follows. In §2, we present a general overview of our prediction method, giving a more detailed description of the extrapolation process in §3. We illustrate the use of our technique with two examples in §4. In §5, we review related work and conclude in §6 with an outline of our planned work.

```
Fortran D Code                          Equivalent SPMD Node Code

REAL a(N)                               REAL a(N/P)
distribute a(BLOCK)

DO i=1,N                                DO i=1,N/P
   a(i) = 1.0                              a(i) = 1.0
END DO                                  END DO
```

Figure 1: Fortran D and SPMD Code Equivalents

2 Performance Scalability Methodology

The performance of a data parallel program when executed on a distributed memory parallel system is a function of many hardware variables, including the integer and floating point processor speeds, cache organization and memory size, and communication link latency and bandwidth. In addition, its performance depends on the efficacy of the compiler translation to SPMD code and the size of the problem being solved. From a performance perspective, one key question is how the code's performance scales as a function of the number of processors and the problem size.

Although the size of most real applications can only be specified with multiple parameters (e.g., the sizes of multiple, multi-dimensional arrays), for simplicity's sake, we consider only codes whose problem size is described by a single parameter N. We believe the techniques we will describe in §3 can easily be generalized to include multiple problem size parameters.

Our goal is to predict application execution time for arbitrary configurations, where we define a *configuration* as a pair (P, N), meaning that an application executes with problem size N on a parallel system with P processors. Our approach uses data gathered during compilation to build a program scalability model that contains the execution cost of each possible program event as a function of P and N. Because not all event costs are determinable at compile time (e.g., data dependent message sizes), the compiler generates SPMD code with embedded performance instrumentation. By executing the instrumented program on one or more base configurations, we can compute the required event costs needed to extrapolate performance to other configurations.

As a simple example, consider a program fragment that initializes a distributed array with a constant. Figure 1 shows the Fortran D source code and a possible translation of that code to equivalent, P processor SPMD code. In addition to creating the SPMD code, the compiler would estimate the cost of this loop as the symbolic expression $\frac{KN}{P}$, where K is the duration of a loop iteration. To determine the value of K, we must execute a version of the program, for a given combination of (P, N), that includes instrumentation to measure the loop duration T_l. Given a value for T_l, we have

$$T_l - \delta_i = \frac{KN}{P}$$

and

$$K = \frac{(T_l - \delta_i)P}{N},$$

where δ_i represents the loop instrumentation overhead, obtained by a series of off-line meta-performance measurements of the instrumentation software. We then predict the non-instrumented loop duration for any new configuration (P', N') as $\frac{KN'}{P'}$.

More generally, we approximate a program's total execution time as the sum of the execution times for all events in the SPMD code. We emphasize that we do not expect the compiler to conduct any analysis of possible program execution paths. Instead, the compiler simply enumerates all the possible SPMD code events that can have non-zero execution time and associates a symbolic expression for execution time with each event. Although this approximation does not consider the potential overlap of distinct events during execution, especially where such events occur on different processors, our experience has been this is not a large source of prediction error for regular, data parallel codes.

3 Prediction Infrastructure

As we noted in §2, the mechanism we use to predict performance is dependent on support from the data parallel compiler. Our current model is based on an integrated compilation and performance analysis system jointly developed by the Rice Fortran D group and the University of Illinois Pablo group [1]. Below, we describe this infrastructure and illustrate its use with a simple example.

3.1 Compiler Scalability Support

In addition to generating SPMD code, the Rice Fortran D compiler also creates a file that describes each program loop, procedure call and communication primitive in the generated code and its relation to the original Fortran D source code. This file, comprising static information about the program, is written in the Pablo SDDF format [2], and contains one record for each program event. As an example, a message passing record specifies the message length (if known at compile time), the identity of the destination processor(s), the scalar or array section carried by the message, and the location of the original Fortran D source code that caused the compiler to synthesize the message passing call.

A detailed description of the SDDF record contents is beyond the scope of this paper; see [1] for details. For performance scalability prediction, the key record types and their fields include the following.

- Loops:
  - loop bounds
  - type of communication pattern inside the loop
  - pointers to other records for events inside the loop
- Procedures:
  - type of communication pattern inside the procedure
  - pointers to other records for events inside the procedure
- Message passing:
  - message length
  - message destination (single node or broadcast)

Both the loop bounds and the message lengths can be expressed as symbolic functions of P, N, or a loop index variable.

In addition to these static performance records, the compiler stores both mapping and semantic information relevant to performance analysis in a static SDDF file. The static records in this file are richly interconnected, allowing one to correlate them in a variety of ways, including how procedures, procedure calls, source loops, data distribution statements, data references, data declarations, and static performance data in the Fortran D source file relate to one another and to the corresponding constructs in the compiler-synthesized SPMD code. This mapping information is used to bind dynamic performance data and our scalability prediction model to the original Fortran D source code.[1]

After compilation, our scalability prediction software can analyze the static records and build a program scalability model. For each SPMD program event, the software symbolically computes the event cost by summing the costs for all its component events. By induction, the aggregate program execution time is derived as the sum of its constituent event times.

Figure 3 shows an overview of the integrated system. The system extends the Fortran D compiler to instrument the compiled code, to record information on compiler analysis and transformations, as well as to export static performance information, uses the Pablo instrumentation software's extension interfaces for capturing dynamic performance data, and incorporates a software toolkit to combine the static and dynamic performance data and relate it to the Fortran D source. In the system, Pablo SDDF serves as a flexible medium of data interchange between the compiler and Pablo. Since static performance data is presented as symbolic expressions of P and N, the system employs a symbolic expression manipulator (e.g., Maple or Mathematica [14]) to derive such expressions. The performance prediction model interacts with data correlation toolkit and symbolic expression manipulator to anaylze performance data and provide performance prediction functionalities.

At this writing, only portions of this mechanism are automated. The Fortran D compiler records mapping and semantic information in an SDDF file and emits SPMD code with embedded calls to the Pablo instrumentation library. At present, the meta-performance benchmarks and the manipulation of symbolic expressions are done manually, though the techniques needed to automate this process are well understood.

3.2 Prediction Details

As an example, suppose that the Fortran D compiler generates the instrumented SPMD code fragment shown in Figure 2. For this fragment, the compiler generates the following static SDDF records.

- Outer loop:
  - Loop bounds: $1, N$
  - Type of communication: upper shift with wrap-around
  - Events inside the loop:
    - inner loop
    - message send
    - message receive

[1] The mapping strategy is discussed in depth in [1].

```
DO i=1, N
   TRACEEvent(LoopEntry)
   DO j=i+1, N
      (... some computation ...)
   END DO
   TRACEEvent(LoopExit)

   TRACEcsend( N/P items, dest=mod(MyNodeID+1, P) )
   TRACEcrecv( N/P items )
END DO
```

Figure 2: Compiler-synthesized SPMD Code Fragment

- Inner loop:
  - Loop bounds: $i+1, N$
  - Type of communication: none
  - Events inside the loop: none
- Message send:
  - Message length: N/P
  - Message destination: single node
- Message receive:
  - Message length: N/P

Scalability prediction software can estimate the cost for each iteration of the outer loop as

$$T_{iteration} = T_{inner-loop} + T_{csend} + T_{crecv}$$

where

- $T_{inner-loop} = (N-i)K_{inner}$
- $T_{csend}(N/P) = \frac{K_{send}N}{P}$
- $T_{crecv} = X_{recv}\left(\frac{N}{P}\right)$

K_{inner} and K_{send} are constants, and the function $X_{recv}(l)$ is the time to receive a message of length l. On most distributed memory systems, $X_{recv}(l)$ takes the form

$$X_{recv}(l) = al + b$$

where a and b are obtained by a least squares fit to measurements of the time to receive a message.

Combining these formulas, we have

$$T_{outer-loop} = \sum_{i=1}^{N} T_{iteration} = \sum_{i=1}^{N} \left[(N-i)K_{inner} + \frac{K_{send}N}{P} + \frac{aN}{P} + b\right]$$

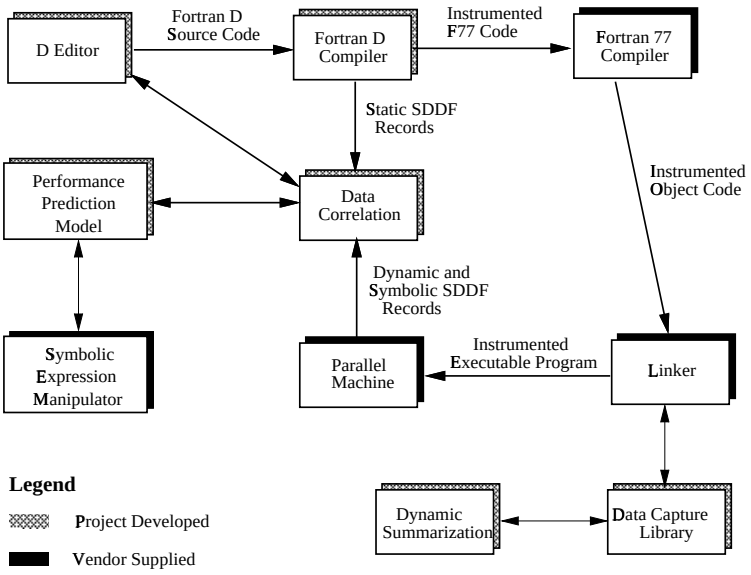


Figure 3: The Integrated Performance Prediction Model

which can be reduced symbolically to

$$T_{outer-loop} = \frac{K_{inner}(N^2 - N)}{2} + \frac{K_{send}N^2}{P} + \frac{aN^2}{P} + bN \tag{1}$$

Note that this presumes the scalability software includes a symbolic expression manipulator (e.g., like Maple or Mathematica).

Except for the unknown constants K_{inner} and K_{send}, equation (1) is a simple function of N and P. To obtain estimates of these constants, we execute the instrumented SPMD code of Figure 2 on some baseline configuration (P_0, N_0). On first glance, one might expect the instrumentation overhead for the code in Figure 2 to be substantial — the instrumentation generates multiple events for each iteration of the outer loop.

To reduce the cost of dynamic instrumentation and the total volume of captured data, we exploit the Pablo instrumentation library's support for real-time data reduction [12]. Rather than recording the generated events for post-mortem analysis, we compute performance metrics as the events are generated. For this example, this summarization would generate only the estimates $L_{inner-loop}$ and S_{csend}, the total time spent in the inner loop and message send function, respectively. Because those two program sections correspond to the first and second terms of equation (1), one can then compute K_{inner} and K_{send} as

$$K_{inner} = \frac{L_{inner-loop} - \delta_i}{\left(\frac{N_0^2 - N_0}{2}\right)}$$

and

$$K_{send} = \frac{S_{csend} - \delta_i}{\left(\frac{N_0^2}{P_0}\right)}$$

where δ_i is the overhead introduced for instrumentation.

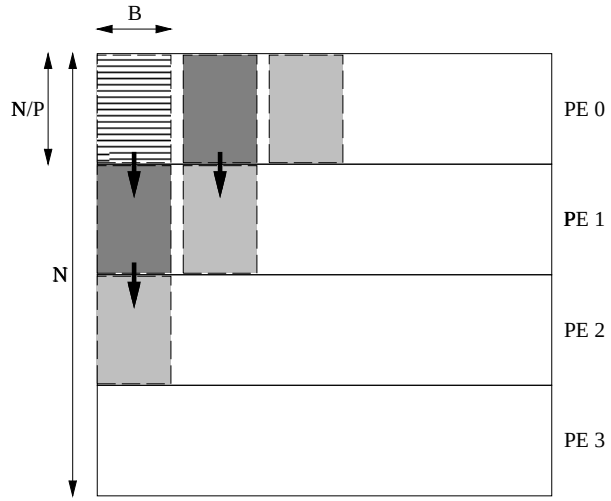


Figure 4: Data distribution and communication for the SOR program

Scalability model construction proceeds similarly for more complex codes. In general, one uses compile-time expressions for the execution times of primitive events (i.e., those without enclosing events) and successively constructs more complex symbolic expressions that represent the execution times of larger code blocks. Dynamic performance data is used to estimate constant values, and the symbolic expressions are simplified via a symbolic expression manipulator.

4 Scalability Prediction Examples

To illustrate the use of our scalability prediction techniques and to assess their accuracy, we present two examples, successive over-relaxation (SOR) and Gaussian elimination, where we predict program execution time for a variety of system and problem sizes. Both example codes were written in Fortran D.

4.1 Successive Over-Relaxation

This first example uses the successive over-relaxation (SOR) method to update the values of a two-dimensional array. The basic updating rule is

$$a'_{i,j} = (1-\omega)a_{i,j} + \frac{\omega}{4}(a'_{i-1,j} + a'_{i,j-1} + a_{i+1,j} + a_{i,j+1}).$$

For our Fortran D implementation of SOR, the array a is distributed among the processors by rows in a blocked fashion. To optimize the execution but still respect the data dependencies in the program, the Fortran D compiler generates communication between processors in a *pipelined* form, using a block of B data items.

Figure 4 shows the data distribution and communication scheme, Figure 5 shows the Fortran D code, and Figure 6 shows the equivalent SPMD code and its associated costs.

```
      double precision a(N,N)          subroutine sor_sub()
      common /buffer/ a                double precision a(N,N)
C     decomposition d(N,N)             common /buffer/ a
C     align a with d
C     distribute d(BLOCK,:)

      DO i = 1, N                      DO j = 2, N-1
         DO j = 1, N                      DO i = 2, N
            a(i,j) = 12.0                    a(i,j) = 0.175 * (a(i-1,j) +...
         END DO                           END DO
      END DO                           END DO

      CALL sor_sub()
      STOP
      END
```

Figure 5: Fortran D SOR Code

4.1.1 SOR Scalability Model

Examining the code in Figure 6 shows that we can estimate the execution time of the SOR program as the sum of the execution times for the initialization loop and the SOR_SUB subroutine

$$T_{SOR} = T_{i\_loop} + T_{sub}$$

where the time for the initialization loop T_{i_loop} is

$$T_{i\_loop} = \sum_{i=1}^{N} \frac{N}{P} K_{j\_loop} = \frac{N^2}{P} K_{j\_loop}$$

and the *sor_sub* subroutine execution time T_{sub} is

$$T_{sub} = T_{shift} + T_{pipe}.$$

In turn, the time to execute the shift operation is

$$T_{shift} = T_{buf} + T_{csend} + T_{crecv} + T_{unbuf}$$

where

- $T_{buf} = K_{buf}(N-2)^2$
- $T_{csend} = K_{send}(N-2)$
- $T_{unbuf} = K_{unbuf}(N-2)$
- $T_{crecv} = X_{recv}(N-2)$

[2] As before, we use notation of the form K_x to denote a constant

```
C   Main program:                                        COSTS:
      double precision  a(0:N/P+1,N)

      DO i=1, N
         DO j=1, N/P                                    KN/P
            a(i,j)=12.0
         END DO
      END DO
      CALL sor_sub()                                    Cost(shift) + Cost(j_loop)
      STOP
      END

C  SOR_SUB subroutine:
      me = mynode()
C  Upper-Shift of a(1,2:N-1):
      IF (me > 0)
         CALL buf(N-2 items)                            K(N-2)
         CALL csend(N-2 items, dest=me-1)               K(N-2)
      END IF
      IF (me < P-1)
         CALL crecv(N-2 items)                          Xrecv(N-2)
         CALL unbuf{N-2 items)                          K(N-2)
      END IF
C  Pipelined loop:
      DO j=2, N-1, B
         IF (me > 0)
            CALL crecv(B items)                         Xrecv(B)
            CALL unbuf(B items)                         K
         ENDIF
         DO k=j, j+B-1                                  B x Cost(i_loop)
            DO i=1, N/P                                 KN/P
               --<< update a(i,k) >>--
            END DO
         END OD
         IF (me < P-1)
            CALL buf(B items)                           K
            CALL csend(B items, dest=me+1)              K
         END IF
      END DO
```

Figure 6: Basic SPMD code structure and event costs for the SOR program.

Constant	Value (Seconds)
K_{j_loop}	1.281×10^{-7}
K_{buf}	3.094×10^{-6}
K_{send}	2.051×10^{-6}
K_{unbuf}	2.790×10^{-6}
$K_{unbuf'}$	7.600×10^{-5}
K_{update}	5.906×10^{-6}
$K_{buf'}$	6.500×10^{-5}
$K_{send'}$	1.080×10^{-4}

Table 1: SOR Scalability Constants (Intel iPSC/860)

Notice that, as we described earlier, we do not consider possible event overlaps in the shift. We will return to the performance implications of this assumption shortly. Finally, we estimate the cost for the pipelined loop with block size B as

$$T_{pipe} = (P + \frac{N}{B} - 1)T_{iteration}$$

where the single iteration time $T_{iteration}$ is given by

$$T_{iteration} = X_{recv}(B) + K_{unbuf'} + \frac{N}{P}K_{update} + K_{buf'} + K_{send'}.$$

Combining all these terms, the scalability prediction for the SOR code becomes

$$\begin{aligned} T_{SOR} = \quad & K_{j\_loop}\frac{N^2}{P} + (K_{buf} + K_{send} + K_{unbuf})(N-2) + X_{recv}(N-2) + \\ & \left(P + \tfrac{N}{B} - 1\right)\left(X_{recv}(B) + K_{unbuf'} + \tfrac{N}{P}K_{update} + K_{buf'} + K_{send'}\right) \end{aligned}$$

4.1.2 Scalability Predictions

Given the scalability model just described, we executed the instrumented SPMD code on an Intel iPSC/860 with four processors and a matrix size of 512. This allowed us to solve for the symbolic constants shown in Table 1.

Using meta-performance benchmarks, we derived the following functions for the time to receive a message:

$$X_{recv}(N-2) = 1.743 \times 10^{-4} + 3.980 \times 10^{-7}(N-2)$$

and

$$X_{recv}(B) = 1.774 \times 10^{-4}.$$

In our predictions, we do not use the observed receive durations. Those receive durations obtained during execution of the instrumented program are, in general, determined by the time spent waiting for a message to arrive, not by the time to actually receive the message data. The best we can do to estimate this later factor is to use the benchmark data above.

Having determined numerical values for all the symbolic constants in the SOR scalability model, we conducted a series of experiments to assess its accuracy. Table 2 shows the observed and predicted execution times, and the relative error associated with each prediction.

In general, the errors are modest to moderate. Recall that our goal is not exact quantitative performance prediction, but rather to provide insights into code scalability and the movement of performance bottlenecks across code fragments as either problem or system size changes. Moreover, the accuracy of predictions is particularly impressive when considering the overhead for software instrumentation. Table 4 shows the SOR execution times are often magnified by a factor of two or more when instrumentation is included. By this standard, the errors are quite acceptable.

Inspecting Table 2 more carefully shows several trends for the SOR code. First, the relative error decreases with increasing problem size. Intuitively, with larger problem sizes, the computation component of the scalability model dominates. Because it is the most accurately estimated model term, the prediction becomes increasingly accurate with larger problems.

Second, the error increases with the number of processors. In this case, our estimates of the symbolic constants were derived from a test execution on only four processors. In general, the larger the disparity between processor baseline and the prediction point, the larger the potential error. We verified that using a baseline configuration with more processors ($P = 8$) indeed produced slightly better predictions. At a minimum, one should use a baseline configuration with enough processors so that the communication behavior in the application code is sufficiently characterized. This can be achieved, in several cases, with four processors.

The majority of the prediction errors in Table 2 are due to the Fortran D compiler's generation of code for the software pipelining of loop execution. In such pipelined loops, a processor waits for a message to arrive from the preceding processor in the pipeline. With the appropriate pipeline granularity, there is little idle time and most of the communication overhead on the sending processor overlaps the communication overhead on the receiving processor. Introducing this overlap in our model is equivalent to considering only the maximum of $X_{recv}(B)$ and $K_{send'}$. To assess the possible benefits of this refinement, we repeated our scalability prediction, obtaining the results in Table 3. Clearly, using information on software pipelines, which is provided by the Fortran D compiler, substantially reduces the prediction error.

The remaining prediction errors are due to application and hardware interactions (e.g., cache behavior). For example, in Table 2, there is an anomaly for four processors and a problem size of 2048. In this case, the problem size and the i860's 8K byte data cache are poorly matched, and nearest neighbor updates cause many cache misses.

In general, the relative contribution of hardware artifacts, compiler code quality and algorithm structure vary across Fortran D codes, and trends in relative prediction errors are strongly dependent on the structure of a particular scalability model. As we shall see in §4.2, different models have different error patterns.

4.2 Gaussian Elimination

As a second example, we consider a Gaussian elimination program (GE), where the data is cyclically distributed across processors by columns. Because our general scalability prediction methodology should now be clear, we omit most details of the derivation, highlighting only those places where the derivation differs from that shown for the SOR code; see [11] for details.

The Fortran D code for this problem is a template. It does not include a back substitution

P	N	Prediction (seconds)	Actual (seconds)	Relative Error (percent)
4	512	0.092	0.080	+15.0
4	1024	0.296	0.288	+2.8
4	2048	1.045	1.359	−23.1
8	512	0.066	0.055	+20.0
8	1024	0.185	0.162	+14.2
8	2048	0.594	0.581	+2.2
16	512	0.055	0.043	+27.9
16	1024	0.132	0.110	+20.0
16	2048	0.371	0.325	+14.2
32	512	0.055	0.041	+34.1
32	1024	0.111	0.087	+27.6
32	2048	0.265	0.221	+19.9
32	4096	0.744	0.656	+13.4

Table 2: Intel iPSC/860 Scalability Predictions (SOR Program)

P	N	Prediction (seconds)	Actual (seconds)	Relative Error (percent)
4	512	0.085	0.080	+6.3
4	1024	0.282	0.288	−2.1
4	2048	1.017	1.359	−25.2
8	512	0.058	0.055	+5.5
8	1024	0.171	0.162	+5.6
8	2048	0.566	0.581	−2.6
16	512	0.047	0.043	+9.3
16	1024	0.117	0.110	+6.4
16	2048	0.342	0.325	+5.2
32	512	0.045	0.041	+9.8
32	1024	0.094	0.087	+8.0
32	2048	0.234	0.221	+5.9
32	4096	0.685	0.656	+4.4

Table 3: Intel iPSC/860 Scalability Predictions with Send/Receive Overlap (SOR Program)

P	N	Instrumented Time (seconds)	Actual Time (seconds)	Overhead (percent)
4	512	0.153	0.080	91
4	1024	0.433	0.288	50
4	2048	1.633	1.359	20
8	512	0.132	0.055	140
8	1024	0.308	0.162	90
8	2048	0.861	0.581	48
16	512	0.126	0.043	193
16	1024	0.263	0.110	139
16	2048	0.636	0.325	96
32	512	0.140	0.041	242
32	1024	0.253	0.087	191
32	2048	0.537	0.221	143
32	4096	1.225	0.656	87

Table 4: Intel iPSC/860 Pablo Instrumentation Overhead (SOR Program)

phase, and it algorithmically initializes the matrix prior to invoking the elimination code. As we shall see, this has several implications for the accuracy of our performance predictions.

4.2.1 Gaussian Elimiation Scalability Model

Unlike the SOR code, which relies on pipelined nearest neighbor communication, the SPMD code synthesized by the Fortran D compiler for Gaussian elimination uses the Intel `csend` primitive to broadcast data. Via a sequence of meta-performance experiments, we measured broadcast costs on the Intel iPSC/860 and fitted the experimental data to a function of the form

$$T_{bcast}(n, P) = c + dnlog_2P$$

where n is the message size, c is -4.647×10^{-4}, and d is 3.482×10^{-7}.

We can estimate the execution time of the GE program as

$$T_{GE} = T_{init} + T_{elim}$$

where T_{init} and T_{elim} are the times for the initialization and elimination loops, respectively. After derivation of the costs for each of the two program sections,[3] and using appropriate values for a and b in the model of X_{recv} corresponding to the time to receive a message on the iPSC/860, as indicated in §3.2, we obtain the following scalability model for Gaussian elimination

$$\begin{aligned} T_{GE} \quad = \quad & K_{init}N^2 + [c + 2d(log_2P)](N-1) + (2a+b)(N-1) + \\ & (K_{loop-1} + K_{loop-2} + K_{buf})\frac{N(N-1)}{2} + \\ & \left[c(N-1) + d\frac{N(N-1)}{2}(log_2P)\right] + \end{aligned} \tag{2}$$

[3] See [11] for details.

Constant	Value (Seconds)
K_{init}	2.044×10^{-6}
K_{reduce}	3.566×10^{-7}
K_{loop-1}	4.902×10^{-7}
K_{loop-2}	2.715×10^{-7}
K_{buf}	1.111×10^{-6}

Table 5: Gaussian Elimination Scalability Constants (Intel iPSC/860)

		Predicted Time (seconds)			Actual Time (seconds)			Relative Error (percent)		
P	N	T_{init}	T_{elim}	T_{GE}	T_{init}	T_{elim}	T_{GE}	init	elim	total
4	512	0.50	4.96	5.46	0.50	4.74	5.24	0.0	+4.6	+4.2
4	1024	2.00	35.89	37.89	2.00	36.17	38.17	0.0	−0.8	−0.7
4	2048	8.02	271.34	279.35	7.83	292.48	300.31	+2.4	−7.2	−7.0
8	512	0.50	3.34	3.84	0.33	3.36	3.69	+51.5	−0.6	+4.1
8	1024	2.00	21.42	23.42	1.31	21.69	23.00	+52.7	−1.2	+1.8
8	2048	8.02	149.66	157.67	5.22	159.94	165.16	+53.6	−6.4	−4.4
16	512	0.50	2.71	3.22	0.24	2.90	3.14	+108.3	−6.6	+2.5
16	1024	2.00	14.92	16.92	0.96	15.28	16.23	+108.3	−2.4	+4.3
16	2048	8.02	91.74	99.76	3.82	96.83	100.65	+109.9	−5.3	−0.9
32	512	0.50	2.58	3.09	0.20	2.89	3.09	+150.0	−10.3	0.0
32	1024	2.00	12.40	14.41	0.78	12.88	13.66	+156.4	−3.7	+5.5
32	2048	8.02	65.71	73.73	3.12	68.37	71.49	+157.1	−3.9	+3.1
32	4096	32.07	390.76	422.82	12.49	413.09	425.58	+156.8	−5.4	−0.6

Table 6: Intel iPSC/860 Scalability Predictions (Gaussian Elimination Program)

$$\left[a\frac{N(N-1)}{2} + b(N-1)\right] + K_{reduce}\left(\frac{N}{6P} - \frac{N^2}{2P} + \frac{N^3}{3P}\right).$$

Executing an instrumented version of the Gaussian elimination program on an Intel iPSC/860 for four processors and N of 1024 yields the numerical values for symbolic constants shown in Table 5

4.2.2 Scalability Predictions

Using equation (2) and the values from Table 5, Table 6 shows scalability predictions for the Gaussian elimination code. The prediction for the elimination loop is quite good, and all errors are 10 percent or less. The prediction for the initialization loop is much less accurate, though this contributes little to the total error because its computational complexity is low compared to the elimination phase.

To understand the reasons for error in the initialization step, consider the Fortran D and equivalent SPMD code for the initialization, shown in Figure 7. We estimate execution time of the outer loop as KN^2 using the loop bounds. However, because of the modulus function, the actual execution time is much closer to $K_1N^2 + K_2N^2/P$. In this case, accurate prediction is only possible

```
C  Fortran D Code

C  distribute a(:,CYCLIC)
   DO j = 1, N
      DO i = 1, N
         x = mod(3125 * init, 65536)
         a(i, j) = (x - 32768.0d0) / 32384.0d0
      END DO
   END DO

C  SPMD Equivalent

   DO j = 1, N
      j$my = ((j - 1) / P) + 1
      DO i = 1, N
         x = mod(3125 * x, 65536)
         IF (MyNodeID .EQ. MOD(j - 1, P)) THEN
            a(i, j$my) = (x - 32768.0d0) / 32384.0d0
         END IF
      END DO
   END DO
```

Figure 7: Initialization for Gaussian Elimination

by exploiting additional compiler information, notably knowledge of the array distribution and the meaning of the node identifier variable. As we continue to build our scalability software, we are working with the Rice Fortran D group to include such capabilities.

5 Related Work

Several groups have developed techniques for predicting the scalability of parallel systems and parallel programs. Carlson *et al* [3] studied the performance of parallel systems using execution profiles, which specify the number of busy processors as a function of time. Using such profiles, the authors identified phases of homogeneous utilization, and characterized program scalability based on the scalability of individual phases. Derivation of the properties for a given phase, however, may require multiple execution with different numbers of processors. In addition, identifying phase transitions can be computationally expensive for long-running codes.

Grama, Gupta and Kumar [6] studied the performance of parallel systems using isoefficiency analysis. For a given algorithm/system combination, they defined the corresponding isoefficiency function as the required growth in the input data set size to maintain the same efficiency, as the number of processors grows. The lower the complexity of the isoefficiency function, the easier it is to achieve scaled speedups with the given algorithm/system pair. Deriving the isoefficiency function requires detailed, extrinsic knowledge of the underlying algorithm and system and would

be difficult to automate.

Sarukkai [15] analyzed the scalability of parallel programs using both static program information and dynamic execution traces. His study is closest in spirit to our approach, as it also automatically builds scalability models for computation and communication sections of an SPMD program. In contrast to our work, Sarukkai's technique does not permit a direct comparison among the costs of individual program sections and assumes the program is already in SPMD form.

Mehra, Schulback and Yan [10] predict performance by modeling message passing programs at varying levels of syntactic detail, using statistical regression techniques to infer the parameters in the model. This modeling process was completely manual. As a next step, Mehra *et al* [9] presented a system for automated modeling, using a grammar-driven approach to describe trace files and estimate numerical parameters in the models by statistical regression. The selection of appropriate terms to include in a model, however, still required human intervention, and the regression procedure needed performance data from numerous executions with different data set sizes and numbers of processors.

Crovella and LeBlanc [4] predicted the performance of parallel programs based on lost cycles analysis. This technique measures overhead categories and fits the measurements to functions that relate overhead to the number of processors and problem size. Though general, the technique presumes the overhead categories are known and exploits no compile-time knowledge of program behavior.

Parashar *et al* [13] presented a framework to predict the performance of HPF/Fortran 90D programs, using an interpretive approach. Using abstractions of HPF source code constructs and benchmark data on the performance of those constructs, they assign costs to application constructs and estimate execution time. This method requires extremely detailed knowledge of the underlying system, and its target is guidance for efficient data distributions.

Fahringer and Zima [5] designed a performance prediction tool named PPPT (Parameter-based Performance Prediction Tool), which analyzes a set of parameters that characterize the behavior of a parallel program, including work distribution, amount of communication and data locality. The tool correlates statically computed information with actual performance measurements to provide the program's performance estimates both to a compilation system and to general users. Such estimates, however, are presented in terms of predicted values for those selected parameters, instead of the execution time of the program or its component sections.

Our work differs from all of these approaches in focusing on the integration of a data parallel compiler with dynamic instrumentation. By combining knowledge of the relation between data parallel source constructs and the compiler-generated SPMD code, with dynamic performance data taken from instrumented execution of the SPMD code, one can develop symbolic expressions describing the scalability of individual data parallel constructs.

6 Conclusions and Future Work

Our preliminary results show that scalability predictions based on compile-time information is a promising complement to traditional performance analysis techniques based solely on experimental data. Our scalability predictions relate compiler-derived program transformation data and dynamic performance data captured during execution of the compiler-synthesized SPMD code. From these two sources we derive a scalability model that consists of symbolic expressions representing the execution complexity of individual program sections. Predicting performance for a program and

architecture configuration is then a simple evaluation of the symbolic expressions. We believe this technique has great promise and is applicable to a variety of code generation models.

Our future work has three major foci. The first is to extend our validation suite to include a much larger suite of codes and range of machine and problem sizes. The second is to exploit even more compile-time information to enhance the quality of our scalability models. In the case of an SPMD code generated by the Fortran D compiler, this will require an analysis of guard expressions (i.e., conditionally executed code). For simple examples, like the initialization loop of the Gaussian elimination code of §4.2, this requires only the detection of special constructs created by the compiler. The third and most ambitious goal is to validate our conjecture that our approach can be easily extended to include other code generation models. At present, we are working with the Portland Group (PGI) to understand the relation between HPF source and compiler-synthesized code. Unlike the Rice Fortran D compiler, the PGI compiler relies heavily on a library of runtime routines for distributed array management.

Acknowledgments

Our thanks to Vikram Adve for useful comments on the work.

References

[1] Adve, V. S., Mellor-Crummey, J., Andersdon, M., Kennedy, K., Wang, J.-C., and Reed, D. A. *An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs*, March 1995. Submitted for publication.

[2] Aydt, R. A. *The Pablo Self-Defining Data Format.* Department of Computer Science, University of Illinois at Urbana-Champaign, Dec. 1994.

[3] Carlson, B. M., Wagner, T. D., Dowdy, L. W., and Worley, P. H. Speedup Properties of Phases in the Execution Profile of Distributed Parallel Programs. Tech. Rep. ORNL/TM-11900, Oak Ridge National Laboratory, August 1992.

[4] Crovella, M. E., and LeBlanc, T. L. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proceedings of Supercomputing'94* (Washington, November 1994), pp. 600–609.

[5] Fahringer, T., and Zima, H. P. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 7th International Conference on Supercomputing* (Tokyo, Japan, July 1993), pp. 207–219.

[6] Grama, A. Y., Gupta, A., and Kumar, V. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology 1*, 3 (August 1993), 12–21.

[7] Hiranandani, S., Kennedy, K., and Tseng, C.-W. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM 35*, 8 (August 1992), 66–80.

[8] Loveman, D. B. High Performance Fortran. *IEEE Parallel & Distributed Technology 1*, 1 (February 1993), 25–42.

[9] Mehra, P., Gower, M., and Bass, M. A. Automated modeling of message-passing programs. In *Proceedings of the ACM/IEEE Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems - MASCOTS'94* (Durham, January 1994), pp. 187–192.

[10] Mehra, P., Schulbach, C. H., and Yan, J. C. A comparison of two model-based performance-prediction techniques for message-passing parallel programs. In *Proceedings of the ACM Conference on Measurement & Modeling of Computer Systems - SIGMETRICS'94* (Nashville, May 1994), pp. 181–190.

[11] Mendes, C. L. *Performance Prediction and Communication Optimization for Multicomputers.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995. In preparation.

[12] Noe, R. J. *Pablo Instrumentation Environment Reference Manual.* Department of Computer Science, University of Illinois at Urbana-Champaign, Dec. 1994.

[13] Parashar, M., Hariri, S., Haupt, T., and Fox, G. C. Interpreting the Performance of HPF/Fortran 90D. In *Proceedings of Supercomputing'94* (Washington, November 1994), pp. 743–752.

[14] Riddle, A. Mathematical power tools. *IEEE Spectrum 31*, 11 (November 1994), 35–47.

[15] Sarukkai, S. R. Scalability Analysis Tools for SPMD Message-Passing Parallel Programs. In *Proceedings of the ACM/IEEE Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems - MASCOTS'94* (Durham, January 1994), pp. 180–186.

A Cost Model and Flow Analysis Based Heuristic for Automatic Data Partitioning

Jodi Tims, University of Pittsburgh

The development of automatic data partitioning heuristics presents a serious challenge in the design of compilers for distributed memory architectures. Analysis techniques must expose variable interrelationships that affect the overall efficiency of a chosen partitioning. In addition, the existence of a cost model that effectively estimates the communication costs and parallelism gains of candidate partitionings is critical to the success of automatic distribution techniques. The cost model must represent the tradeoff between parallelism and communication and reflect communication cost savings that are realizable through optimization techniques. This presentation introduces an automatic partitioning heuristic that addresses these issues. The technique utilizes data flow information rather than code structure to identify sets of array variables that will be partitioned together.

Predicting Contention in Distributed-Memory Machines

Arjan J.C. van Gemund, Technical University Delft, The Netherlands

A compile-time prediction technique is outlined that yields low-cost, highly parameterized performance models, to be used during the initial optimization loops in parallel system design. Aimed to provide an acceptable accuracy across a large parameter search space the approach is based on extending conventional static analysis with asymptotic queuing analysis in order to predict the effect of contention. In this paper we report on the accuracy of the prediction method compared to simulation results as well as to measurement results on a distributed-memory machine.

Predicting Contention in Distributed-Memory Machines

Arjan J.C. van Gemund
a.vgemund@et.tudelft.nl

Department of Electrical Engineering
Delft University of Technology
P.O.Box 5031, NL-2600 GA Delft, The Netherlands

Abstract

A compile-time prediction technique is outlined that yields low-cost, highly symbolic performance models, to be used during the initial optimization loops in parallel system design. Aimed to provide an acceptable accuracy across a large parameter search space the approach is based on extending conventional static analysis with asymptotic queueing analysis in order to account for potentially dominating effects of resource contention. In this paper we report on the accuracy of the prediction method when compared to simulation results as well as compared to actual measurement results on a distributed-memory machine.

1 Introduction

In the performance prediction of parallel systems many approaches exist which represent a specific trade-off between analysis accuracy and cost. Although compile-time techniques entail a sacrifice in accuracy when compared to e.g., simulation, this loss may be acceptable during the first phases of system design in view of the low cost and high level of parameterization that can be achieved. While the cost issue plays a prominent role in the development towards automatic system optimization, the symbolic nature of the models enables an efficient parameter (e.g., scalability) study based on a one-only developed model. Current static approaches have already shown promise in the area of compile-time as well as user-level mapping optimization [7, 12, 52]. Whereas static techniques focus on task synchronization they do not account for resource contention except for a number of *ad hoc* approaches. Queueing for resources, however, can easily degrade performance by orders of magnitude compared to the optimistic outcome of a simple static technique. Clearly, a static approach which sustains a minimum accuracy across a large parameter (search) space would be of value.

The following example illustrates the possible result when problem partitioning is performed using a traditional static technique that does not account for (link) contention.

Example 1 Consider a simple farmer-worker parallelization of an arithmetic reduction algorithm (e.g., summing a list of numbers [27]) on a P-processor distributed-memory machine. In this example (and later on in this paper) we consider a Parsytec GCel T800 transputer system[1] using either $P = 2$, $P = 3$, or $P = 4$ nodes. The nodes are (purposely) organized in a linear array $0 - 1 - 2 - 3$ in which node 0 has been designated to act as the farmer (of course, no human would select this arangement but an automatic mapper might come across it during its search).

Let the problem size be $N = 4 \times 10^6$ bytes. Let the computational work load involved be 24 CPU seconds. A part of the problem (N_w) is sent to $P - 1$ workers in parallel. One part ($N - (P-1)N_w$) is processed directly on the farmer. This part is larger than the other partitions to minimize completion time. Communication delays for the scalar return values are ignored. For $P = 4$ the distribution is shown in Figure 1. Let the communication delay be given by the traditional bandwidth model $T^c = 0.9 \times 10^{-6} l$ (s) where l denotes the data size in bytes. (In view of the large transfers the startup delay has been ignored for simplicity.) Based on

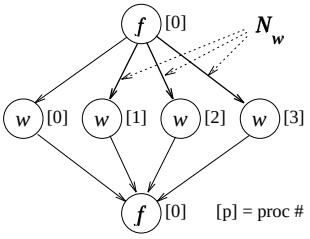


Figure 1: Task distribution for $P = 4$

this model an optimal data distribution can be determined by calculating the optimal value of N_w. For each of these distributions Table 1 lists the execution time predictions T^t as well as the actual execution times T^m measured on the transputer system. Whereas a parti-

[1]Kindly made available by the Interdisciplinary Center for Computer-based Complex systems research Amsterdam (IC^{3}A).

P	N_w (B)	T^t (s)	T^m (s)
1	0.00×10^6	2.4	2.4
2	1.14×10^6	1.7	1.8
3	0.89×10^6	1.3	2.1
4	0.73×10^6	1.1	2.9

Table 1: Distributions based on traditional model

tioning procedure based on traditional static prediction would select $P_{opt} = 4$ and predict $T^t = 1.1$, in reality $T^m = 2.9$ due to the large degree of link contention that is involved. □

Aimed to overcome the fundamental lack of prediction robustness of traditional static techniques a compile-time method has been proposed, which, at the same low cost, approximately accounts for resource contention. The analysis method is defined in terms of a formalism called PAMELA (PerformAnce ModEling LAnguage [15]). Both (parameterized) program and machine are modeled in terms of separate PAMELA submodels, which, when combined by substitution, results in a model L of the complete system. Rather than simulating (executing) L to obtain the execution time estimate T, L is compiled into a fully parameterized performance model [16] which yields an alternative, lower bound T^l at much lower cost. This analytic alternative method is based on a simple calculus which has been coined serialization analysis.

Although initial case studies have already indicated that T^l is reasonably sharp when compared to the simulation result T, the exact nature of the sacrifice in accuracy has not yet been explicitly addressed. In this paper we study the properties of T^l for random series-parallel (SP) graphs, both through a series of 1200 simulation experiments as well as through actual measurements on the transputer mesh. It is shown that for a large class of task systems the average error of T^l relative to T is limited within a factor of 2. Moreover, it is shown that for many graphs the gravity of this error can be predicted. Although the inaccuracy prohibts parameter fine-tuning, the technique does enable a first-order ranking (as well as an absolute execution time estimate) between alternative designs (e.g., partitionings) which may easily entail very large performance differences (as illustrated even by the very small example above).

The rest of the paper is organized as follows. For the sake of completeness, in Section 2 we briefly present the formalism, the compile-time calculus, as well as an account of the related work in the performance prediction field. In Section 3 we study the nature of the average relative prediction error based on the simulation experiments. In Section 4 we report on the case study involving the transputer mesh measurements. The paper is concluded in Section 5.

2 Pamela

2.1 Formalism

Basically, PAMELA is an imperative simulation formalism extended with a compile-time calculus. In the following we briefly describe the language features that are relevant to this paper. Apart from the conditional control flow operators **if-else** and **while**, PAMELA includes binary (infix) operators to describe sequentialism (i.e., '**;**') and fork/join parallelism (i.e., '||'). The parallel operator has implicit barrier semantics. While the above constructs permit the construction of SP models, in order to express non-SP structures, **wait** and **signal** operators are provided for explicit task synchronization. **wait**(c) blocks the invoking process(es) until condition c is asserted by a call to **signal**(c).

Work is described by the **use** construct, like in **use**(s, τ), in which the invoking process (task) exclusively acquires service from server s for τ units time (excluding possible queueing delay). In the sequel we will often refer to servers as (active) *resources*. A resource s has a multiplicity, denoted $|s|$ which will be assumed 1 unless stated otherwise. The service time may be deterministic or stochastic. In the compile-time calculus, however, only deterministic (or mean) values will be considered. In order to describe simultaneous resource possession, a generalization of the **use** construct is provided, defined as **using** (s) L, in which each statement of L is executed under the condition that s is acquired. Thus resource access can be nested. An example of this will be discussed later on. Like in queueing networks, it is convenient to define an infinite-server ρ such that $|\rho| = \infty$. Instead of **use**(ρ, τ) we will simply write **delay**(τ). Finally, replication is described by the reduction operators **seq** and **par**, defined by

$$\begin{aligned} \textbf{seq}\ (i = a, b)\ L_i &= L_a\ ;\ \ldots\ ;\ L_b \\ \textbf{par}\ (i = a, b)\ L_i &= L_a \parallel \ldots \parallel L_b \end{aligned}$$

Corresponding to the formal approach towards model construction and analysis, we write a PAMELA model according to a (process-algebraic) equation syntax, which implies a simple substitution semantics.

Consider the following PAMELA model

$$L = \textbf{par}\ (i = 1, 100)\ \{\textbf{delay}(10)\ ;\ \textbf{use}(r, 1)\}$$

where $|r| = 1$ and in which the temporal parameters are deterministic. The resulting (simulated) execution time is given by $T = 110$ due to the serialization of the **use** statements. Note, that traditional static analysis yields $T^t = 11$. Our analytic technique predicts a lower bound $T^l = 100$ as will be shown later on.

As can be seen from the choice of synchronization mechanism the programming paradigm in PAMELA is *procedure-oriented* rather than *message-oriented* [4]. The choice for this paradigm corresponds to the

material-oriented [28] approach towards the modeling of concurrent systems. Whereas a *machine-oriented* approach (using a message-passing formalism) is generally associated with ease of modeling, material-oriented modeling offers more ease of *analysis* [15].

Example 2 Consider a machine repair model (MRM) in which P clients either spend a mean time τ_l on local processing, or request service from a server s ($|s| = 1$), with mean service time τ_s, with a total cycle count of N iterations (unlike steady-state analysis, in our approach we require models to terminate). Both times are assumed to be exponentially distributed. The PAMELA model of the MRM is specified by

$$
\begin{array}{l}
L = \textbf{par}\ (i = 1, P) \\
\qquad \textbf{seq}\ (i = 1, N)\ \{ \\
\qquad\quad \textbf{delay}(\tau_l); \\
\qquad\quad \textbf{use}(s, \tau_s) \\
\qquad \}
\end{array}
$$

in which the exclusive service is expressed by the **use** operation applied to resource s which represents the server. The example illustrates the material-oriented modeling approach in which the server is modeled by a passive construct. In the machine-oriented approach, the server would map to a separate process which synchronizes through message-passing. □

Example 3 Consider the pipelined processing of N data sets involving an M unit pipeline (e.g., vector unit, packet-switched communication, software pipeline). In a machine-oriented paradigm, each unit would map to a process which would synchronously receive a data set, process it, and send it to the next unit. Thus the natural programming paradigm is message-oriented. In the PAMELA approach, the model is material-oriented in which the entire computational process (involving M stages) is expressed for each data set. The result is a *contention model* in which each data process is executed in parallel and contends for each unit in the course of its computation. The PAMELA model is given by

$$L = \textbf{par}\ (i = 1, N)\ \textbf{seq}\ (m = 1, M)\ \textbf{use}(u_m, \tau_m)$$

where u_m denotes the resource that represents unit m, and τ_m denotes the associated processing time. Note, that, although the absolute order in which data is processed is left undetermined, the performance prediction remains valid (no data-dependencies). The above model correctly predicts both startup delay as well as the bandwidth of the pipelined system. □

Example 4 Consider the following SPMD program which scales a sparse N element vector $\underline{v}$ using a simple P-node scalar multiprocessor, i.e.,

```
for (i = 0; i < B(p); i++)
    if (v[f(p,i)] != 0)
        v[f(p,i)] *= alpha;
```

where $p = 0 \ldots P-1$ denotes the processor index and $B(p)$ and $f(p,i)$ denote local loop bound and index function, respectively (based on partitioning scheme). Let the multiprocessor be modeled in terms of the instructions `move` (shared memory load/store) and `flop` (local floating point operation including register traffic). Given this interface, the PAMELA model of the parallel program is given by the expression

$$
\begin{array}{l}
L = \textbf{par}\ (p = 0, P-1) \\
\qquad \textbf{seq}\ (i = 0, B(p)-1)\ \{ \\
\qquad\quad move(v + f(p,i)); \\
\qquad\quad \textbf{if}\ (v[f(p,i)] \neq 0)\ \{ \\
\qquad\qquad flop; \\
\qquad\qquad move(v + f(p,i)) \\
\qquad\quad \} \\
\qquad \}
\end{array}
$$

where the **par** section accounts for the SPMD execution. Many details like multiprocessing overhead are ignored for simplicity (`alpha` is assumed to be already loaded in register).

Consider a simple machine architecture based on a non-pended bus (i.e., circuit-switched) bus b (single bus, hence $|b| = 1$). Then the PAMELA machine model is given by the equations

$$
\begin{array}{rcl}
flop & = & \textbf{delay}(\tau_f) \\
move(a) & = & \textbf{use}(b, \tau_m)
\end{array}
$$

where τ_f denotes the effective floating point instruction time, and τ_m denotes the effective memory load/store time. Note, that in this architecture the actual memory address (a) is irrelevant. Consequently, the application model is given by

$$
\begin{array}{l}
L = \textbf{par}\ (p = 0, P-1) \\
\qquad \textbf{seq}\ (i = 0, B(p)-1)\ \{ \\
\qquad\quad \textbf{use}(b, \tau_m); \\
\qquad\quad \textbf{if}\ (c(p,i))\ \{ \\
\qquad\qquad \textbf{delay}(\tau_f); \\
\qquad\qquad \textbf{use}(b, \tau_m) \\
\qquad\quad \} \\
\qquad \}
\end{array}
$$

where $c(p,i) = (v[f(p,i)] \neq 0)$ denotes the data dependency. □

As the emphasis in this paper is on the *analysis*, the use of PAMELA in *modeling* shared and distributed-memory programs and (vector) machines is discussed elsewhere [15]. More examples including the analysis of conditional constructs can be found in [17]. Note, that PAMELA's operators essentially enable the same modeling accuracy when compared to (hybrid) task graph/queueing approaches, while hardware and software are modeled in terms of one formalism. Through the natural expression of data-dependent control flow sometimes even a larger degree of modeling detail is possible.

Being an imperative formalism, a PAMELA model L can be directly executed (simulated [18]). This will yield a prediction T which accounts for conditional control flow, task synchronization, and contention with relatively high precision, provided of course, that the model is detailed enough and sufficient simulation runs are performed given the (generally) non-deterministic control flow and contention involved. In view of the huge computation costs we use this evaluation mode only in order to validate our analytic approach.

2.2 Serialization Analysis

As mentioned before, our analytic approach is based on a lower bound approximation of contention integrated within a static compile-time task graph analysis scheme. As state space analysis is avoided altogether, we can describe the estimation approach in terms of a symbolic calculus which allows the resulting performance model to be fully parameterized. In the following we briefly summarize the analysis. More details can be found in [17].

Although in many cases conditional control flow will be accounted for in terms of e.g., (effective) branch probabilities or statement frequencies, the following transformation, i.e.,

$$\textbf{if } (c)\ \textbf{use}(r,\tau) \rightarrow \textbf{use}(r,[c]\tau)$$

shows in principle how conditional control flow is handled in the symbolic analysis. The construct $[\ldots]$: $\{false, true\} \rightarrow \{0,1\}$ denotes Iverson's operator [23] defined by $[false] = 0$ and $[true] = 1$. Since a PAMELA model is block-structured (i.e, no **gotos**), the above transformation can be applied recursively, eventually yielding a model without conditionals. Thus, if needed, control parameters can be mapped onto the resulting performance model.

As in conventional compile-time approaches, task synchronization analysis is based on a critical path analysis in which we ignore the effects of contention (i.e., each **use** statement is interpreted as if it were a **delay** statement). Let φ denote the isomorphism between a PAMELA computation and the corresponding (execution) time computation in the time domain. Then $\varphi(L)$ denotes the estimated execution time of L. We informally introduce the isomorphism in terms of the task graph representations of both computations. Consider a PAMELA model L represented by the task graph G. Let g_i denote a task in G equivalent to the PAMELA statement $\textbf{delay}(\tau_i)$. Let π_i denote the set of predecessors of g_i. Let $\varphi(L)$ be represented by the task graph H. Then H is isomorphic to G where h_i is given by the computation

$$t_i = \max_{j \in \pi_i} t_j + \tau_i$$

where t_i is an intermediate output variable of task h_i. The 'max' term accounts for the delay due to task synchronization while τ_i accounts for the actual work. Thus L maps to a set of equations of the above form. Let g_e denote the terminal task. Then the numeric result for $\varphi(L)$ is given by t_e. In terms of the PAMELA operators '**;**' and '||', it follows

$$\begin{aligned}
\varphi(L_1\ ;\ldots;\ L_N) &= \varphi(L_1)+\ldots+\varphi(L_N),\\
\varphi(L_1 \parallel \ldots \parallel L_N) &= \max(\varphi(L_1),\ldots,\varphi(L_N)),\\
\varphi(\textbf{delay}(\tau)) &= \tau,\\
\varphi(\textbf{use}(r,\tau)) &= \tau
\end{aligned}$$

which forms the basis for traditional SP reduction.

The effect of resource contention is approximated by a simple *lower bound* analysis based on a computation of the total service demand as shown in the following. Let $\underline{\delta}(L) = (\delta_1,\ldots,\delta_M)$ denote the total service demand vector of L where M is the total number of resources involved and δ_m denotes the total service demand on resource r_m. For convenience we will write $\delta_m(L)$ to denote the m-th element of $\underline{\delta}(L)$. Clearly,

$$\begin{aligned}
\underline{\delta}(L_1\ ;\ldots;\ L_N) &= \underline{\delta}(L_1)+\ldots+\underline{\delta}(L_N),\\
\underline{\delta}(L_1 \parallel \ldots \parallel L_N) &= \underline{\delta}(L_1)+\ldots+\underline{\delta}(L_N),\\
\underline{\delta}(\textbf{use}(r_m,\tau)) &= \tau\underline{e}^m
\end{aligned}$$

where $\underline{e}^m = (0,\ldots,0,1,0,\ldots,0)$ is the M-dimensional unit vector in the m direction, and addition and multiplication are defined element-wise. Let ω denote the lower bound on the execution time of L due to the fact that each access to a resource is at least serialized. Then

$$\omega(L) = \max_{m=1\ldots M} \left\lceil \frac{\delta_m(L)}{|r_m|} \right\rceil$$

Combining the lower bound due to contention (ω) with the estimation due to task synchronization (φ) it follows that the lower bound on T is predicted by

$$T^l(L) = \max(\varphi(L),\omega(L)) \qquad (1)$$

While Eq. 1 applies to basic parallel sections, for general (possibly non-SP) models the following recursion provides a sharper bound as will be illustrated in Example 8.

$$\begin{aligned}
T^l(L_1\ ;\ldots;\ L_N) &= T^l(L_1)+\ldots+T^l(L_N),\\
T^l(L_1 \parallel \ldots \parallel L_N) &= \max(T^l(L_1),.,T^l(L_N),\omega(L)),\\
T^l(L) &= \max(\varphi(L),\omega(L)) \qquad (2)
\end{aligned}$$

Note, that conventional compile-time analysis disregards ω while queueing analysis (partially) disregards φ. Due to the fact that, in addition to the critical path, we account for the serialization due to contention, we have coined this lower bound approach *serialization analysis*. Like conventional compile-time analysis, for SP models serialization analysis has only a linear complexity in the (symbolic) size of the model.

Example 5 Recall the MRM in Example 2. It follows

$$\begin{aligned}\varphi &= \max_{p=1\ldots P}\sum_{i=1}^{N}(\tau_l+\tau_s) = N(\tau_l+\tau_s)\\ \omega &= \sum_{p=1}^{P}\sum_{i=1}^{N}\tau_s = PN\tau_s\end{aligned}$$

By Eq. 1 (or Eq. 2) it follows $T^l = N\max(\tau_l+\tau_s, P\tau_s)$. Unlike conventional compile-time analysis T^l accounts for the additional queueing delay when s is saturated. The above analysis yields the same result as asymptotic bound analysis in queueing theory [30]. Let R denote the response time and let $Z = \tau_l$ denote the think time. Then the mean cycle time $R+Z$ equals φ/N for $P \ll P^*$ and ω/N for $P \gg P^*$, where the saturation point $P^* = (\tau_s+\tau_l)/\tau_s$ denotes the crossover between the asymptotes. □

Example 6 Recall the pipeline model in Example 3 given by

$$L = \textbf{par}\ (i=1,N)\ \textbf{seq}\ (m=1,M)\ \textbf{use}(u_m,\tau_m)$$

Let a pipeline unit take τ_a on average, and let c denote the slowest pipeline unit. It follows

$$\varphi = M\tau_a,\quad \omega = N\tau_c$$

Thus Eq. 1 yields $T^l = \max(M\tau_a, N\tau_c)$. Indeed, T^l is a lower bound as $T = M\tau_a+(N-1)\tau_c$. However, the relative deviation is negligible for cases where either φ dominates ($M \gg N$) or where ω dominates ($N \gg M$). The maximum deviation occurs for a *balanced* system (i.e., equal resource demands, $\tau_c = \tau_a = \tau$) when $N = M$). In this case, $T^l = N\tau$ whereas $T = (2N-1)\tau$. Hence, the worst case relative deviation is a factor 2. Note, however, that this can only occur for balanced systems which are precisely in the cut-off region between no saturation and full saturation. □

Example 7 Recall the SPMD computation in Example 4. It follows

$$\begin{aligned}\varphi &= \max_{p=0\ldots P-1}\sum_{i=0}^{B(p)-1}(\tau_m+[c(p,i)](\tau_m+\tau_f))\\ \omega &= \sum_{p=0}^{P-1}\sum_{i=0}^{B(p)-1}(1+[c(p,i)])\tau_m\end{aligned}$$

which yields the prediction T^l. In the interest of efficiency we consider two possible reductions of T^l. First, let the density of $\underline{v}$ be uniform and measured to be given by d. As a result of the reduction

$$\sum_{i=0}^{B(p)-1}[c(p,i)] = dB(p)$$

it follows

$$\begin{aligned}\varphi &= \max_{p=0\ldots P-1}B(p)\,(\tau_m+d(\tau_m+\tau_f))\\ \omega &= PB(1+d)\tau_m = N(1+d)\tau_m\end{aligned}$$

which reduces the complexity of T^l from $O(N)$ to $O(P)$. Furthermore, under the assumption of a standard block or cyclic partitioning scheme an $O(1)$ complexity model results, i.e.,

$$T^l = \max(\left\lceil\frac{N}{P}\right\rceil(\tau_m+d(\tau_m+\tau_f)), N(1+d)\tau_m)$$

Note, that the actual mean value of T^l is slightly higher due to the variance of each node process (offset logarithmically in P [29]). In practice, however, the error is limited due the fact that for larger P the ω term will dominate, which yields the correct asymptote (central limit theorem). □

Example 8 In order to demonstrate the vital importance of Eq. 2, consider the following model, i.e.,

$$L = \textbf{seq}\ (i=1,N)\ \textbf{par}\ (p=1,P)\ \textbf{use}(r_i,\tau)$$

in which resource usage is non-uniformly distributed over the length of the entire computation. While Eq. 1 yields $T^l = \max(P\tau, N\tau)$, Eq. 2 yields $T^l = \sum_{i=1}^{N}\max(P\tau,\tau) = NP\tau$. Thus applying Eq. 2 (i.e., applying Eq. 1 to *each* parallel section instead of only once) improves the bound by as much as a factor N. □

The choice to account for contention in terms of a lower bound model is based on two reasons. In contrast to the computation of a (sharp) upper bound, a sharp lower bound can be computed at the same cost as conventional compile-time techniques. Moreover, as illustrated by Example 5 and 6, the lower bound model correctly predicts the average execution time of systems that are either contention-free or fully saturated. This property which conforms to the aim of our approximate approach is further elaborated in Section 3.

The introduction of contention analysis in static prediction techniques is not entirely new. A lower bound comparable to Eq. 1 has been used by Allen *et al.* [3] to account for the limited number of processors in compile-time prediction of dynamically scheduled task graphs. Recently, an improved lower bound has been described by Jain and Rajaraman [24] to predict the lower bound for multiprocessor optimal schedules. Comparable to our recursive approach (Eq. 2) they improve the sharpness of the basic bound given by Eq. 1 by applying the analysis to separate task graph layers (cf. Example 8). Their results apply to dynamic scheduling of M equal processors (i.e., one resource *cpu* with $|cpu| = M$), whereas our generalization applies without any constraints on resource types or demands.

2.3 Related Work

In order to put our approach into perspective, in this section we review some of the many interesting approaches to the analysis of the completion time of parallel systems. We characterize the work in terms of the representation formalism used (e.g., task graphs, queueing nets, Petri nets, languages) as well as how the three major aspects of system behavior, i.e., task synchronization, contention, and conditional (data-dependent) control flow, are analyzed.

Many approaches focus on the analysis of task synchronization, typically represented in terms of a task graph. In order to account for the non-determinism of conditional control flow and contention, tasks are often associated with stochastic (often exponentially distributed) delays. Many methods are defined for SP graphs as the state space analysis can be efficiently implemented in terms of SP reduction. Gelenbe *et al.* [14] describe a method to compute the overall probability density function for general task time distributions. Sahner and Trivedi [42] describe a method to compute this function for task time distributions that has an exponential polynomial form. Other approaches focus on the approximation of the completion time for non-SP graphs. Lester [31] describes a method to compute the overall distribution by modeling task distributions by their generating function (z-transform). Yazici-Pekergin and Vincent [53] consider bounds on the mean completion time for general i.i.d. (independent identically distributed) task times. Hartleb and Mertsiotakis [20] describe a bounding analysis by approximating a graph in terms of SP versions with the use of heuristics. Aimed to keep the state space analysis manageable, Sötz [48] describes an approximation method based on using deterministic and/or exponentially distributed task time variables only.

Compile-time approaches are typically based on analyzing task graphs with deterministic (scalar) task times. While the above stochastic approaches inherently account for the non-determinism due to conditional control flow, in the compile-time approach this aspect is either accounted for through simple reductions based on the mean condition probabilities, or by simply determining the instruction frequencies for a specific data set (e.g., based on sequential profiles [12]). Since the variance of task times is ignored, the potential increase of the mean completion time due to (barrier) synchronizations is not accounted for[2]. As a result of the predominant data parallel nature of parallel programs, compile-time analysis is typically based on SP reduction which implies a solution cost only linear in the size of the program source. Approaches to this effect have been described by Atapattu and Gannon [6], Balasundaram *et al.* [7], Lester [31] (as an alternative to the above cited analysis), Polychronopoulos and Banerjee [40], Sarkar [43], So *et al.* [47], and Wang [52]. Atapattu and Gannon take a partially parametric approach. The symbolic approach taken by Clement and Quinn [10] is also based on reduction of a static graph model. Unlike the above approaches, however, the work loads are determined using multiple linear regression techniques.

While the above approaches account for task synchronization and conditional control flow (especially in the case of stochastic graphs), they disregard an explicit analysis of contention due to the inherent limitations of the task graph representation. While some of the deterministic approaches include an approximate analysis of (memory and/or network) contention (e.g., [6, 52]), the above probabilistic approaches only account for contention in terms of the non-deterministic task distributions (or additional "contention" nodes [34]). Consequently, in both approaches the task parameters may need to be adjusted when choosing different machine architectures.

Also in the probabilistic field, alternative approaches have been described which combine a stochastic task graph representation at program level with a queueing network which accounts for the contention at machine level (hierarchical decomposition). Thomasian and Bay [50] describe a method based on analyzing the Markov chain derived from the graph using the results of queueing analysis for the state transition rates. A comparable approach has been described by Kapelnikov *et al.* [26]. In order to decrease the state space analysis complexity, Mak and Lundstrom [33] describe a method which is optimized for SP graphs (cf. SP reduction in the earlier approaches). A somewhat different approach is taken by Jonkers *et al.* [25] in which non-SP task graphs with deterministic task times are analyzed. The underlying premise is that the actual variance at task level does not always justify the necessity of a high-variance distribution like the exponential distribution, an argument which recently has been put forward by Adve and Vernon [1] (we will return to this point, later on).

A number of approaches have been described which are based on the use of stochastic Petri nets (exponential transition rates) for modeling both program and machine (e.g., see [13, 51]). As a result of the high modeling power of Petri nets [38], both task synchronization and contention can be expressed. Aimed to reduce the underlying state space analysis complexity, Ajmone Marsan *et al.* [2] describe an extension with immediate transitions to model control flow or activities with negligible time delays. Other approaches to realize state space reduction are described by Plateau [39], Buchholz [9], and Siegle [45].

While the above representation formalisms are essentially different from the von Neumann systems un-

[2] Parallel tasks with equal mean times entail additional synchronization delay at a mutual barrier due to their variance. This phenomenon has been investigated for a wide class of distributions by Kruskal and Weiss [29].

der study, concurrent (simulation) languages offer a natural means to express both synchronization types as well as conditional control flow. A typical example is the process-oriented simulation language CSIM described by Schwetman [44], which combines both a procedure-oriented and message-passing programming paradigm [4]. Due to the above advantages a vast number of prediction approaches based on simulation modeling have been described (e.g., [36, 49, 35, 41, 46]). Unlike the earlier approaches, however, simulation languages typically lack an explicit compile-time *analytic* tradition. A recent alternative approach, is the use of stochastic process algebras, described by Götz *et al.* [19], which combines the advantages of language-based model construction with an underlying state space analysis.

As mentioned earlier, our approach to performance prediction is primarily motivated by low cost and high level of parameterization, rather than (absolute) prediction accuracy (the purpose of this paper, of course, is to assess the sacrifice in this respect). In order to achieve a high level of parametrization, the approach is language-based with an analysis method which is conducive to the generation of symbolic performance models. Effectively, this rules out any (numeric) state space analysis method, narrowing the options down to a static method comparable to current compile-time approaches. Unlike the above task graph approaches, however, we aim to account for contention in a natural way in order to provide the robustness needed in view of the large parameter range covered by a largely symbolic performance model. The basic premise of our approach is the following. If the control flow for a representative data set can be accounted for in terms of mean task durations, a parametric analysis method is indeed possible when contention can be analytically approximated. As shown by Adve and Vernon [1], task time variance is usually limited, which implies that for many systems the analysis error due to the assumption of deterministic service times (mean values) will be acceptable in view of the overall approximation (i.e., in contrast to fully stochastic approaches). Since task synchronization is accurately accounted for by critical path analysis, the accuracy of the method will be largely determined by the approximation error with respect to the analysis of contention. In this paper we show that the average error is limited even when a simple lower bound analysis is used.

3 Average Accuracy

3.1 Introduction

In theory, for highly concurrent models which involve many resources the absolute range between the lower bound T^l and the upper bound T^u within which T lies is quite considerable. This implies that for models that feature coarse grain (task level) contention, the prediction error of T^l relative to T may also be large due to the high variance of T. Examples are applications featuring large critical software sections and statically (and poorly) scheduled systems which may exhibit quite disappointing execution time results when compared with predictions based on an average or a lower bound[3]. In [17] a study is presented into the range between T^u and T^l for various model classes.

In this paper, we study the effects of fine-grain (subtask-level) contention in terms of the *average* of T since for practical systems, typically featuring frequent, random resource access, the variance in T is much less compared to the above situation. As illustrated by Example 5 and Example 6, for models in which the resource demand is reasonably uniform during the entire computation (i.e., in contrast to models such as in Example 8), T^l approaches the mean value of T either when $\varphi \gg \omega$ (critical path dominates) or when $\varphi \ll \omega$ (queueing dominates). Thus, in many cases the average error of the analytic prediction is quite acceptable (as experiments will show later on). As already mentioned in Example 5 the choice of the lower bound as a practical estimate is also inspired by similarities between the execution of L and interactive queueing systems. Although, formally, the resemblance is extremely remote it is interesting to relate the lower bound approach to the asymptotic bound analysis of an (operationally) comparable interactive queueing system[4]. If we define Z as the think time [30], D as the total service demand and D_{max} as the service demand at the bottleneck device, we can interpret φ as the horizontal cycle time asymptote $D+Z$ (Z accounts for task synchronization delay), while ω corresponds to the ND_{max} asymptote. The largest deviation occurs at the saturation point, where $D+Z = ND_{max}$.

As a result of the above observations we propose to use an operational metric called "serialization index" which characterizes the degree of contention within a system. The metric is defined by

$$\theta = \log\left(\frac{\omega}{\varphi}\right) \tag{3}$$

The use of θ is to characterize a model as to the the likelyhood of T^l being an accurate prediction. For models with large $|\theta|$ the average accuracy of T^l is expected to be better than for models where $|\theta| \approx 0$.

[3] In the case of a multicomputer application we observed that a different schedule due to a simple program loop reversal reduced communication time by a factor 2 due to the elimination of (link) contention. Similar observations are reported by Culler *et al.* [11].

[4] Note, that the comparison is purely intuitive as we disregard many details, e.g., the fact that each task should map to a unique job class; possible transient phases like startup and shutdown are ignored; the task graph should be cyclic in order to have steady state execution, etc.

3.2 Experiments

In this section we report on an experiment involving 1000+ random SP models in which the predictions T^l are compared to the simulation results T. The models are generated such that the θ values lie around the (worst case) region of interest ($|\theta| \approx 0$). Apart from the fact, that many computations of interest are SP structured[5], the choice for SP models is also motivated by the fact that it enables an evaluation of the improvement of Eq. 2 on the accuracy compared to Eq. 1. Each model comprises $N = 100$ tasks while the number of resources involved varies from $M = 2 \ldots 150$. The graphs are generated by a simple algorithm which iteratively adds a new task t_i to a random selected task t_j within the graphs generated up to then (j is determined in each iteration). The probability that t_i is placed in series or parallel with t_j is determined by an input parameter, denoted s. Each task t_i is characterized by a unique service demand vector $\underline{\delta_i} = (\delta_{i,1}, \ldots, \delta_{i,M})$ in which each element is i.i.d. uniformly over $[0, 1]$. Thus, balanced systems are generated (on average). Experiments have verified that this choice indeed provides the worst case with respect to the accuracy of T^l. Each resource m is accessed multiple times based on the existence of some deterministic service time τ. Thus each task executes $\delta_{i,m}/\tau$ accesses to resource m. The order in which the resources are visited is random. In order to minimize simulation time (many models are simulated), τ is chosen such that the mean of T does not deviate significantly from results for $\tau \to \infty$ (in practice, values in the order of 1 % of the largest service demand $\delta_{i,m}$ (i.e., ≈ 100 visits) have been found to suffice[6]). As N is fixed ($N = 100$), the parameters M and s determine the (mean) θ value of the generated models. As φ is proportional to M, large values of M will generate models with a negative θ. For low s, however, many parallel tasks are created on average which has a positive influence on θ.

Figure 2 shows the ratio T^l/T based on 1200 random models exhibiting θ values ranging from $-2 < \theta < 2$. Both the prediction ratios based on Eq. 1 (α) and Eq. 2 are shown (β). Each data point of both series of 120 points represents an average value based on 10 random draws in order to reduce noise. The results clearly show a high correlation between (α, β) and θ in which the deviation from unity is indeed maximal for models which exhibit $\theta = 0$. Thus, for random graphs the diagnostic value of the operational parameter θ appears to be quite significant, especially when considering the

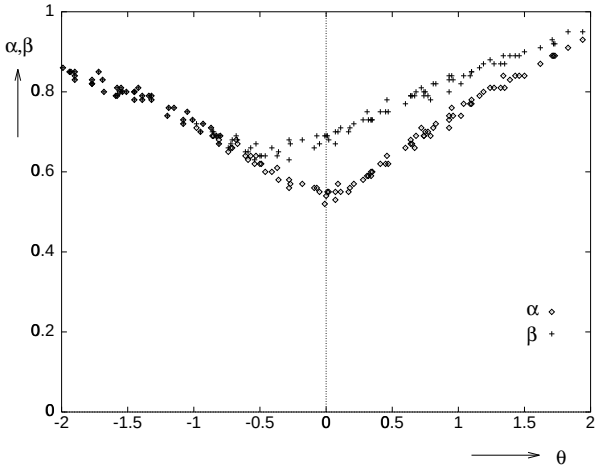


Figure 2: T^l accuracy for random models ($N = 100$)

fact that two graphs with comparable θ values usually have quite a different structure. While the essential necessity of Eq. 2 has already been demonstrated (cf. Example 8), even for the models with uniform resource demand (in time) as produced by the random generator, its application still yields an improvement for models with highly parallel subsections (e.g., $\theta > 0$). In the following we will only consider α. In the above experiments the models are generated for $s = 0.1$ with M varying from $M = 2$ ($\theta \approx 2$) to $M = 150$ ($\theta \approx -2$). Models with $\theta \approx 0$ are generated for $M = 20$. For each value of M the variance of θ is approximately 0.05 which accounts for the reasonably continuous plot. Although no extensive experiments have yet been performed for different values of N, initial measurements indicate that the α curves tend to be more 'v'-shaped for small N, corresponding to the fact that the scale of θ is still somewhat dependent on the problem size. However, $N = 100$ also appears to be quite representative for larger models[7] as well. Additional measurements (described in [17]) indicate that the *minimum* value of α at $\theta = 0$, i.e., α^*, highly correlates with M. For instance, each of the following set of parameter tuples,

$$(N, M, s) \in \{(30, 8, 0.1), (100, 8, 0.3), (300, 8, 0.5)\}$$

generates models with $|\theta| \approx 0$ that yield $\alpha^* \approx .6$ on average. The results show the existence of an asymptote for large M given by $\alpha^* \approx 0.5$. Consequently, for the random SP models we consider the worst case average deviation of T^l relative to T is limited to a factor 2.

Again, it is tempting to compare this upper bound on the (mean) deviation with the result from asymptotic bounding analysis of interactive queueing systems. For instance, consider the MVA recursion for an M

[5] Note, that the application range of PAMELA SP models is essentially greater than just SP task graphs. For instance, pipelining can be expressed in terms of a parallel section of contending tasks (cf. Example 3).

[6] The justification is that for large visit counts the change in task-level variance proves to be small (which is in agreement with the results of Adve and Vernon [1]). This phenomenon has been observed for both exponential and deterministic service times.

[7] For $N = 10$ the range of interest is $-0.5 < \theta < 0.5$, whereas for $N \geq 1000$ the range is still around $-2 < \theta < 2$. A fully normalized version of θ is beyond the scope of this study.

server balanced system [54] with total service demand D, given by

$$R(N) = D + \frac{D}{M}\,\frac{R(N-1)}{R(N-1)+Z}\,(N-1) \qquad (4)$$

where $R(N)$ denotes the response time as a function of the number of jobs in the system (N). The mean cycle time (comparable to T) is given by $C(N) = R(N) + Z$. From Eq. 4 it follows that for small N the slope of $R(N)$ is (still) less[8] than D/M. Consequently, at the saturation point $N = N^*$, for which the deviation between $C(N)$ and its lower bound $C^l = D + Z$ is the largest, it holds $C(N^*) < C^l + (D/M)N^*$. With $N^* = (D+Z)/(D/M)$ it follows $C(N^*) < 2C^l$, which corresponds to the lower bound on α^*. Generating models with large M also implies a large D. In terms of the analogy this implies a decreasing Z. Indeed, from Eq. 4 it is easily seen that $\lim_{Z\to 0} C(N^*) = 2C^l$ which is in agreement with our measurements.

4 Case Study

4.1 Introduction

In order to validate our technique as well as the observations in the previous section we present a case study in which the measured execution times of 15 random programs on a distributed-memory machine are compared with our predictions based on both simulation and our analytic technique. Each program involves a data flow-style execution of a random SP task graph on a 16 processor mesh partition of the transputer system mentioned in the introduction. The task graphs are generated by the same random generator as used in the previous section. Again, $N = 100$. Each task t_i executes a simple loop kernel with a random loop count given by w_i. A task t_i is randomly mapped onto a processor p_i according to a uniform distribution between 1 and $P = 16$. Thus, on average, 100/16 multiple tasks are mapped onto the same processor. Each task is executed by a separate (lightweight) thread scheduled dynamically by a node's run-time kernel. In order to enable true data flow execution, after each task has executed, the (same) produced data set (l_i bytes) is asynchronously sent to each successor task (thread) except when a successor resides locally. A typical example of this type of application is described in [32] where the task graphs represent sparse finite element computations in structural analysis.

Due to the dynamic approach towards task computation and communication, the case study (intentionally) provides an excellent example of the added value of serialization analysis compared to conventional static prediction techniques. While static analysis inherently ignores the additional delay incurred by tasks sharing a processor, our approach naturally accounts for this delay by modeling task execution in terms of "processor contention". Apart from this, the use of non-blocking communication introduces the possibility of link contention as multiple task communications may share the communication link(s) at the same time. Again, conventional static analysis makes no provision to account for the additional queueing delay, which may easily dominate performance (as will be shown). In our aim just to demonstrate the impact contention analysis may have, we simply consider coarse grain task execution where each task entails a large amount of computation ($O(10^6)$ floating point operations) as well as communication ($O(10^6)$ byte transfers). Hence, without loss of generality we can simply concentrate on computational and communication *bandwidths* rather than startup times (and other sources of overhead), which simplifies the discussion.

The message-passing interface used for the implementation is based on the "virtual link" service, which provides a dedicated logical channel between a sender and receiver task. The virtual link topology needed to connect predecessor and successor tasks is created in the prologue of the actual program. (Thus link setup times are not measured.) As mentioned earlier the communication mode selected is "asynchronous" (non-blocking) without additional buffer copy. In terms of the message-passing system interface the communication is implemented using `ARecv`, `ASync`, and `ASend` calls [37]. A more detailed description of the architecture of the SPMD application appears in [17].

4.2 Computation Model

Let G denote the task graph to be executed, consisting of tasks $t_i, i = 1, \ldots, N$. Let L denote the PAMELA model of the SPMD program instantiated by data set G. Then L is given by starting with a graph topologically similar to G where each task i specifies a computation model $comp(i)$. In addition, every arc between a task i and j is expanded by a communication model $xfer(i, j, l)$ which accounts for the communication (l is message length in bytes) induced by that arc (discussed at length elsewhere). Since G is an SP graph the resulting program model L is also an SP graph which implies that L can be expressed in terms of one single (possibly complex) PAMELA expression. As a result, L can be directly compiled into a single expression T^l based on the application of Eq. 2. For instance, consider the following 6-tasks graph

$$G = t_1;\ \textbf{par}\ (i = 2, 5)\ t_i;\ t_6$$

The PAMELA model of the SPMD program is given by the expression

[8] An accurate analysis of the balanced upper bound is given by Zahorjan *et al.* [54]. However, for our purpose the above analysis suffices.

$L = comp(1);$
par $(i = 2, 5)$ {
$xfer(1, i, l_1);$
$comp(i);$
$xfer(i, 6, l_i)$
};
$comp(6)$

In [17] a formalization[9] is given of the construction of the program model L.

The *comp* model represents the actual task computation. In our aim to evaluate the prediction accuracy in the presence of processor and network contention, we refrain from modeling the local computation in detail and simply measure the loop kernel as a whole. (Hence, the actual form of the kernel is arbitrary.) For the amount of work we consider ($w_i = 10^4 \ldots 10^6$ loops) the execution time increases linearly with w_i according to 6.1 μs per iteration (startup time negligible). The execution time including queueing delay due to processor sharing is expressed by the following (contention) model

$$comp(i) = \textbf{seq}\ (k = 1, 6.1w_i/\tau_c)\ \textbf{use}(p_i, \tau_c)$$

expressed in μs where τ_c denotes the basic CPU time slice. In the coefficient the (small) effect of multithreading overhead is automatically accounted for since during the measurement the above code is run as a thread. In the above example let $\underline{p} = (0, 0, 1, 0, 1, 0)$. When the *xfer* model is ignored (discussed at length in the next section) Eq. 2 yields the symbolic prediction $T^l = 6.1(w_1 + \max(w_2 + w_4, w_3 + w_5) + w_6)$. Indeed, for large computations and zero communication, the above prediction not only equals T (i.e., the simulation result) but also closely matches the actual execution time measured (within a few percents, as shown later on). Note, that for this example conventional static analysis may already yield an error up to 100 %.

4.3 Communication Model

In conventional static techniques a **delay** model is used to predict the transfer delay (e.g., [5, 8, 21]). Although precise for *isolated* transfers, these models generally do not account for additional queueing delay induced by concurrent traffic on the intermediate (sending, forwarding, receiving) links and nodes. In the following we develop a transfer contention model which provides a first-order bandwidth approximation for *simultaneous* communications. Although the model is still based on a number of (simplifying) assumptions the predictions prove to be quite accurate as is shown later on. (In view of the large prediction error of a **delay** model in the presence of network load even a rough contention model would mean a considerable improvement.)

In contrast to the task indices used above, in the following we will consider $xfer(s, r, l)$ where s and r denote sender and receiver *node*, respectively (i.e., $i = p_s$, $j = p_r$). The virtual link service of the transputer system is based on a multiplexing scheme in which each 120 bytes of the message is packetized. Each packet is statically routed through the mesh in a pipelined fashion. Pending an upgrade based on the T9000, the virtual link service is still emulated by the node kernel. Consequently, each transfer not only induces work load at all the hardware links in the circuit but also at software level on each forwarding node.

Instead of using **delay** constructs the additional knowledge concerning the potential contention delay is expressed in terms of **use** constructs which refer to the limited number of services available (cf. the **use** model for a task computation instead of a **delay** model). With each physical link between neighboring transputers we will associate a *service complex* comprising a subsystem of physical (e.g., DMAs at both link ends) and/or semi-logical (software servers at both ends) resources. Without any loss of generality, we project the service complex at the receiving node of each link, as shown in Figure 3. In the following we consider the communi-

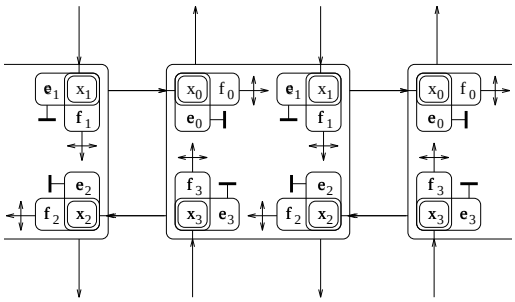


Figure 3: Message-passing service model of the T800 transputer mesh

cation system at the packet level which is the smallest level of granularity with respect to resource sharing. Although the service complex at each link comprises several software/hardware components, it can be modeled as to provide two services at the packet level which are subsequently denoted e and f (see Figure 3).

The first service e represents the reception service at the packet destination involving the exclusive transfer of one packet across the link (duration $\tau_x = 108$ μs [17]), including the software overhead at both ends (e.g., moving, handshaking). The second service f represents the forwarding service (including intermediate byte storage and protocol overhead) required for a packet destined for a different node. Con-

[9] Note, that it might seem obvious to use G for the performance model of the SPMD program (which executes G). Formally, however, we must deal with the fact, that L must represent the SPMD *program* of which the performance is measured, not its data *input*. Hence, we adopt this formal line of reasoning.

sequently, compared to e, f includes additional routing/forwarding work load (duration $\tau_y = 73$ μs). In general, a packet transfer from node $s = n_1$ to $r = n_K$ will require forwarding at $n_2 \ldots n_{K-1}$ (according to static x-y routing[10]) and one reception service at n_K. Both services are based on an underlying service, represented by the resource x which represents the basic link service that has to be shared. Consequently, e and f are *logical* resources (kernel servers) sharing the underlying link service.

Typical for the PAMELA methodology, we use a material-oriented approach to model packet propagation, as this approach is conducive to our analytic method. In the following we model the transfers on a packet level rather than on a byte level. However, the error is negligible in view of the large data transfers we will consider in the case study. In order to model the pipelined packet propagation we model the entire transfer as a parallel operation (cf. Example 3.). Let $n_k = s \ldots r$ denote the index of the K nodes involved in the pipeline route. Then the PAMELA model is given by the contention model

$$
\begin{aligned}
xfer(s,r,l) = \textbf{par}\ & (i = 1, l/120)\ \{ \\
& \textbf{seq}\ (k = 2, K-1) \\
& \quad \textbf{using}\ (f_{n_k})\ \{ \\
& \qquad \textbf{use}(x_{n_k}, \tau_x); \\
& \qquad \textbf{delay}(\tau_y) \\
& \quad \} \\
& \textbf{using}\ (e_r) \\
& \qquad \textbf{use}(x_r, \tau_x) \\
\}
\end{aligned}
$$

Note, that this model ignores startup delay and approximates the work load in terms of an integer number of packets. However, for large data communications this model suffices to accurately capture the effective bandwidth degradation when many virtual links are simultaneously active.

The contention model has been validated for many types of concurrent communications (equal message lengths) as well as random patterns (as discussed later on). Table 2 shows a few typical results for (10^6 byte) data transfers involving only the first row of the mesh (nodes 0, 1, 2, and 3). In the table only the most significant digits are displayed for the ease of interpretation. The nodes that are simultaneously sending are expressed by the $\underline{s}$ vector, while the receivers are expressed by the $\underline{r}$ vector. Each pair (s_n, r_n) corresponds to one communication. Apart from the measured value T^m and the simulation result T the lower bound prediction T^l as well as the traditional static prediction T^t are listed. The results for T show that the *xfer* model is quite accurate for a first-order approximation. Only in a very few situations a limited deviation is measured

[10] First, $|(s/4) - (r/4)|$ nodes in the sender column are traversed after which $|(s \bmod 4) - (r \bmod 4)|$ nodes in the receiver row are traversed (in terms of row-major node numbers).

$\underline{i}$	$\underline{j}$	T^m	T	T^l	T^t
(0)	(1)	0.9	0.9	0.9	0.9
(0)	(2)	1.5	1.5	1.5	1.5
(0,0)	(1,1)	1.8	1.8	1.8	0.9
(0,0)	(2,2)	3.0	3.0	3.0	1.5
(0,0)	(1,2)	1.8	1.8	1.8	1.5
(0,0,0)	(1,1,2)	2.7	2.7	2.7	1.5
(0,0,0)	(1,2,2)	3.3	3.3	3.0	1.5
(0,0,0,0,0,0)	(1,1,1,2,2,2)	6.0	5.4	5.4	1.5

Table 2: Simulation and analytic results for 10^6 byte concurrent communications (s)

(cf. last row). This optimistic prediction is due to the precise packet scheduling which is left undetermined in the PAMELA model. In contrast to practice, this nondeterminism sometimes leads to a bit more optimistic prediction of the possible schedule than the actual implementation.

The table also shows that, despite the high value of $\theta(xfer)$ (*xfer* is expressed as a contention model) the lower bound prediction T^l not always equals T (cf. second last row). This is due to the fact that the nested resource usage prevents a straightforward application of the calculus presented earlier. In fact, T^l has been derived from an approximate version of *xfer*. In this model the effect of simultaneous resource possession is approximated in terms of parallel section of constituent resource requests according to the generic transformation [15], $\textbf{using}\ (r_1)\ \textbf{use}(r_2, \tau) \rightarrow \textbf{use}(r_1, \tau) \parallel \textbf{use}(r_2, \tau)$. Thus the simplified model used for the compilation of T^l is given by

$$
\begin{aligned}
xfer(s,r,l) = \textbf{par}\ & (i = 1, l/120)\ \{ \\
& \textbf{seq}\ (k = 2, K-1)\ \{ \\
& \quad \textbf{use}(f_{n_k}, \tau_x + \tau_y) \parallel \\
& \quad \textbf{use}(x_{n_k}, \tau_x) \\
& \} \\
& \textbf{use}(x_r, \tau_x) \\
\}
\end{aligned}
$$

Effectively, this model accounts for the fact that the communication bandwidth is determined by the largest work load on either an f or an x (or e) server. Note, that this leads to a somewhat less tight bound because the approximate model involves less synchronization constraints than the earlier model.

As mentioned earlier, the above approach is merely intended in order to derive a first-order bandwidth approximation. Hence, many phenomena are not accounted for, such as the influence of concurrent communication in reverse direction on the (duplex) link performance, as well as the communication overhead on the CPU. Both phenomena may introduce errors in the order of 20 % as shown by measurements. The second phenomenon influences the *computation* performance,

G	T^m	T	T^t	θ	α	β
G_1	118.7	114.7	25.9	1.08	0.66	0.73
G_2	93.9	92.5	21.2	0.85	0.53	0.62
G_3	95.6	92.8	25.8	0.76	0.60	0.68
G_4	94.1	87.4	31.5	0.37	0.53	0.70
G_5	73.4	70.9	30.3	0.27	0.56	0.66
G_6	105.8	103.9	58.3	-0.19	0.55	0.55
G_7	98.4	87.0	47.4	-0.28	0.54	0.60
G_8	89.2	87.1	52.5	-0.35	0.61	0.68
G_9	87.6	84.4	65.7	-0.73	0.78	0.78
G_{10}	109.5	106.4	79.5	-0.91	0.75	0.75
G_{11}	141.2	138.4	107.6	-1.23	0.78	0.80
G_{12}	149.5	144.8	125.0	-1.51	0.87	0.87
G_{13}	165.9	163.2	140.2	-1.62	0.86	0.86
G_{14}	172.3	171.0	165.4	-1.70	0.96	0.96
G_{15}	188.6	186.8	174.3	-1.74	0.94	0.94

Table 3: Measurements vs. predictions (s)

G	T_f^m	T_f	T_f^t	T_c^m	T_c	T_c^t
G_1	38.7	38.9	21.3	94.7	88.7	5.3
G_2	46.3	46.5	17.2	67.3	61.9	4.3
G_3	36.1	36.2	22.8	77.6	69.8	4.2
G_4	48.3	48.1	27.4	73.2	64.4	5.8
G_5	54.9	55.3	25.4	32.3	30.0	4.8
G_6	79.1	79.4	47.6	42.5	39.4	10.6
G_7	58.2	58.4	39.7	49.2	42.7	9.8
G_8	63.7	64.0	44.2	36.2	34.0	11.2
G_9	75.7	76.2	56.4	19.5	17.5	10.2
G_{10}	82.6	82.8	65.4	49.3	45.6	16.4
G_{11}	114.8	115.4	91.2	37.3	33.2	19.6
G_{12}	113.0	113.5	103.3	44.8	42.7	22.2
G_{13}	130.8	131.1	119.3	41.2	38.5	23.5
G_{14}	138.2	139.1	136.3	44.2	42.5	28.9
G_{15}	145.8	146.5	145.0	47.5	46.1	33.2

Table 4: Results for f-mode and c-mode (s)

rather than communication performance (communication tasks by the CPU are run at high priority while computation threads have low priority). The first phenomenon, however, directly relates to *communication* performance. Duplex communication essentially degrades communication performance. Each data transfer on a link induces acknowledgement traffic on the reverse link (2 bits per 11 bit datum [22]). Consequently, when data transfers are performed concurrently in both directions, effective link bandwidth drops with approximately 18 %. For instance, a concurrent communication $(0,1) \rightarrow (1,0)$ of 10^6 bytes indeed takes about 1.06 s instead of 0.9 s.

4.4 Results

In this section we present the measurement results for the execution of 15 random SP graphs $G_1 \ldots G_{15}$ on the 4×4 transputer mesh. Each graph comprises $N = 100$ tasks which are randomly i.i.d. uniformly over the 16 processors. The computational work load w_i is i.i.d. uniformly over $[10^4, 10^6]$ (loops) which corresponds to an average total problem size of 305 s. In order for the communication to have a significant impact, the data size sent by each task to its successors is also i.i.d. uniformly over $[10^4, 10^6]$ (bytes) which corresponds to an average communication delay between 0.9 s and 1.5 s per *isolated* transfer. As in the simulation experiment, the graphs are generated to cover the (worst case) region in the neighborhood of $\theta = 0$ by varying s from $s = 0.1$ (parallel graphs, corresponding to low G indices) to $s = 0.5$ (more sequential graphs). Table 3 summarizes the main results for each of the 15 programs. T^m denotes the measured execution time (s). T denotes the simulation result (s) of the corresponding PAMELA model L (within 1 % variation). As mentioned in the previous section, the PAMELA model is based on a version of G in which each *xfer* model is given by the approximate contention model (the difference with the exact model is practically negligible). The result T^l of serialization analysis is represented in terms of θ, α, and β which are defined as before. The total number of resources involved in the simulation and analysis is $M = 144$ (P processors, $4P$ link, and $4P$ forwarding services). The T^t value (i.e., φ) has been included to demonstrate the (severe) prediction error of traditional static analysis.

The results show that the performance of the SPMD program is indeed captured by the PAMELA model with reasonable accuracy. On average, T under-estimates T^m with about 4 % which is entirely due to the fact that the communication model ignores the effects of reverse communication and the additional CPU load (as discussed before; this will also be shown in the next table). The results for α and β indeed show that the average prediction error of serialization analysis is limited to a factor 2 while for models with $|\theta| \gg 0$ the predictions tend to approach the simulation results. As expected, for relatively parallel graphs, the β values tend to be somewhat better than the α values. The increase in accuracy for positive θ, however, appears to be somewhat less when compared to the simulation results in Section 3. This phenomenon is caused by the fact that for many mappings parallel communications involving the same resource may occur at relatively concentrated points in time. Thus, the (link) resource usage is not always uniformly distributed over the length of the entire computation. The effect on the accuracy of T^l is similar to the phenomenon described in Example 8. An elaborate explanation appears in [17].

In order to validate the PAMELA model in more detail, each of the 15 graphs is also executed under a mode f in which all communication (except task synchronization) is switched off ($\underline{l} = \underline{0}$), and a mode c, in which

all computation has been switched off ($\underline{w} = \underline{0}$). Thus each original measurement T^m is complemented by a communication-less version T_f^m and a computation-less version T_c^m, representing both ends of the spectrum. Table 4 shows a comparison of T^m, T, and T^t for both execution modes (T^l behavior comparable to earlier results). The results show that the inaccuracy of T is indeed due to the communication model since the average accuracy of the PAMELA model for f-mode execution lies well within 1 %. In contrast, the c-mode model under-estimates communication delay by 8 % on average. Note, that the results automatically demonstrate the general validity of the (approximate) communication model for random concurrent communication patterns. Finally, note that for a high communication density the difference between T_c^t and T_c (as well as T_c^l) is quite spectacular.

5 Conclusion

We have outlined a simple compile-time prediction technique that yields an analytical estimate of the execution time of a parallel system. The approach is inspired by the need for low cost, highly parametric models to be used during the initial (automatic) optimization loops in parallel system synthesis. Aimed to provide an acceptable accuracy with respect to the influence of resource contention, an analytical lower bound estimate T^l is derived by combining critical path analysis typical for compile-time methods, with asymptotic bounding analysis from queueing theory, integrated within one unified formalism.

In this paper we have investigated the accuracy of T^l compared to T based on simulation studies as well as on experimental data. To this purpose, we have studied the average deviation between T^l and T both through a simulation study involving 1200 random SP graphs as well as through measurements of 15 random SP graphs executed on a 16 node transputer mesh. Whereas the simulation to obtain T requires hundreds of seconds on a typical workstation, T^l evaluates in a split second. Results of both case studies, show that the worst-case penalty for large random task systems is a mere 50 % under-estimation on average. Moreover, it is shown that for a large class of random parallel task systems this error can be reasonably predicted by the metric θ, which is compiled as a side result of T^l.

To the best of our knowledge this approach towards the integration of contention analysis within a symbolic performance modeling technique has not yet been described. Although the accuracy of compile-time analysis is inherently limited, the predictability of the average error allows the estimation technique to be applied in performance engineering approaches in which a quick, first-order analysis of design decisions is appropriate.

Apart from the above motivation, our approach is inspired by the observation that in many cases a deterministic approach to task-level synchronization analysis is quite acceptable in comparison with the more accurate stochastic models [1]. Hence, the estimation of (non-deterministic) conditional control flow has not been explicitly addressed. Typical for compile-time approaches, a mean value approach is used, e.g., based on dataset-specific profile information. For cases where task-level variance cannot be ignored, our compile-time scheme can be extended to account for the additional synchronization delay based on analytic approximations (e.g. [29]). For instance, consider a critical path analysis involving (μ, σ) tuples rather than just scalar (μ) values.

Future research includes validation of our technique for non-SP models (initial experiments suggest comparable results), the application to more programs and machines, as well as an enhancement to the current experimental environment in the form of a PAMELA compiler [16] of which prototype development has recently commenced [18].

Acknowledgement

It is a pleasure to acknowledge my indebtness to Professor Gerard L. Reijns for supporting this research.

References

[1] V.S. Adve and M.K. Vernon, "The influence of random delays on parallel execution times," in *Proc. 1993 ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, May 1993, pp. 61–73.

[2] M. Ajmone Marsan, G. Balbo, and G. Conte, "A class of Generalized Stochastic Petri Nets for the performance analysis of multiprocessor systems," *ACM Tr. on Comp. Syst.*, vol. 2, May 1984, pp. 93–122.

[3] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar, "A framework for determining useful parallelism," in *Proc. 1988 Int. Conf. Parallel Proc.*, Aug. 1988, pp. 207–215.

[4] G.R. Andrews and F.B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 266, no. 24, 1983, pp. 132–145.

[5] M. Annaratone, C. Pommerell, and R. Rühl, "Interprocessor communication and performance in distributed-memory parallel processors," in *Proc. 16th Symp. on Comp. Archit.*, May 1989, pp. 315–324.

[6] D. Atapattu and D. Gannon, "Building analytical models into an interactive prediction tool," in *Proc. Supercomputing '89*, ACM, 1989, pp. 521–530.

[7] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partioning decisions," in *Proc. 3rd ACM SIGPLAN Symp. on PPoPP*, Apr. 1991.

[8] L. Bomans and D. Roose, "Benchmarking the iPSC/2 hypercube multiprocessor," *Concurrency–Practice and Experience*, vol. 1, 1989, pp. 3–18.

[9] P. Buchholz, "Hierarchical Markovian models: Symmetries and reduction," in *Proc. 6th Int. Conf. Modelling Techniques and Tools for Comp. Perf. Eval.*, Edinburgh, U.K., Sept. 1992.

[10] M.J. Clement and M.J. Quinn, "Multivariate statistical techniques for parallel performance prediction," in *Proc. 28th Hawaii Int. Conf. on System Sciences, Vol. II*, IEEE, Jan. 1995, pp. 446–455.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," in *Proc. 4th ACM SIGPLAN Symposium on PPoPP*, May 1993, pp. 1–12.

[12] T. Fahringer and H.P. Zima, "A static parameter-based performance prediction tool for parallel programs," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 207–219.

[13] A. Ferscha, "A Petri net approach for performance oriented parallel program design," *Journal of Parallel and Distributed Computing*, vol. 15, 1992, pp. 188–206.

[14] E. Gelenbe, E. Montagne, R. Suros, and C.M. Woodside, "Performance of block-structured parallel programs," in *Parallel Algorithms and Architectures* (M. Cosnard *et al.*, eds.), North-Holland, 1986, pp. 127–138.

[15] A.J.C. van Gemund, "Performance prediction of parallel processing systems: The PAMELA methodology," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 318–327.

[16] A.J.C. van Gemund, "Compiling performance models from parallel programs," in *Proc. 8th ACM Int. Conf. on Supercomputing*, Manchester, July 1994, pp. 303–312.

[17] A.J.C. van Gemund, "On the accuracy of compile-time performance prediction," Tech. Rep. 1-68340-44(1994)02, Delft University of Technology, Delft, The Netherlands, Sept. 1994.

[18] A.J.C. van Gemund, "The PAMELA run-time library version 1.0," Tech. Rep. 1-68340-44(1994)03, Delft University of Technology, Delft, The Netherlands, Apr. 1994.

[19] N. Götz, U. Herzog, and M. Rettelbach, "Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras," in *Performance Evaluation of Computer and Communication Systems (Combined Tutorial Proceedings SIGMETRICS'93 and PERFORMANCE'93, LNCS 729)* (L. Donatiello and R. Nelson, eds.), Springer, 1993.

[20] F. Hartleb and V. Mertsiotakis, "Bounds for the mean runtime of parallel programs," in *Proc. 6th Int. Conf. Modelling Techniques and Tools for Comp. Perf. Eval.*, Edinburgh, UK, Sept. 1992, pp. 197–210.

[21] R.W. Hockney, "Performance parameters and benchmarking of supercomputers," *Parallel Computing*, vol. 17, 1991, pp. 1111–1130.

[22] INMOS Limited, *The Transputer Databook*, 1989. Doc. No. 72 TRN 203 01.

[23] K.E. Iverson, *A Programming Language*. Wiley, 1962.

[24] K.K. Jain and V. Rajaraman, "Lower and upper bounds on time for multiprocessor optimal schedules," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, Aug. 1994, pp. 879–886.

[25] H. Jonkers, A.J.C. van Gemund, and G.L. Reijns, "A probabilistic approach to parallel system performance modelling," in *Proc. 28th Hawaii Int. Conf. on System Sciences, Vol. II*, IEEE, Jan. 1995, pp. 412–421.

[26] A. Kapelnikov, R.R. Muntz, and M.D. Ercegovac, "A modeling methodology for the analysis of concurrent systems and computations," *Journal of Parallel and Distributed Computing*, vol. 6, 1989, pp. 568–597.

[27] A.H. Karp, "Programming for parallelism," *Computer*, May 1987, pp. 43–57.

[28] W. Kreutzer, *System simulation, programming styles and languages*. Addison-Wesley, 1986.

[29] C.P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. 11, Oct. 1985, pp. 1001–1016.

[30] E.D. Lazowska *et al.*, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[31] B.P. Lester, "A system for computing the speedup of parallel programs," in *Proc. 1986 Int. Conf. Parallel Proc.*, IEEE, Aug. 1986, pp. 145–152.

[32] H.X. Lin and H.J. Sips, "Parallel direct solution of large sparse systems in finite element computations," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 261–270.

[33] V.W. Mak and S.F. Lundstrom, "Predicting performance of parallel computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, July 1990, pp. 257–270.

[34] A.D. Maloney, V. Mertsiotakis, and A. Quick, "Automatic scalability analysis of parallel programs based on modeling techniques," in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 794) (G. Haring and G. Kotsis, eds.), Berlin, Springer-Verlag, May 1994, pp. 139–158.

[35] P. Mehra, C.H. Schulbach, and J.C. Yan, "A comparison of two model-based performance-prediction techniques for message-passing parallel programs," in *Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, Nashville, May 1994, pp. 181–189.

[36] K.M. Nichols and J.T. Edmark, "Modeling multicomputer systems with PARET," *Computer*, May 1988, pp. 39–48.

[37] Parsytec Computer GmbH, *Parix release 1.2 software documentation*, Mar. 1993.

[38] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[39] B. Plateau, "On the stochastic structure of parallelism and synchronization models for distributed algorithms," in *Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, Austin, Aug. 1985, pp. 147–154.

[40] C.D. Polychronopoulos and U. Banerjee, "Speedup bounds and processor allocation for parallel programs on multiprocessors," in *Proc. 1986 Int. Conf. Parallel Proc.*, Aug. 1986, pp. 961–968.

[41] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood, "The wisconsin wind tunnel: Virtual prototyping of parallel computers," in *Proc. 1993 ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, May 1993, pp. 48–60.

[42] R.A. Sahner and K.S. Trivedi, "SPADE: A tool for performance and reliability evaluation," in *Modelling Techniques and Tools for Performance Analysis '85* (N. Abu El Ata, ed.), Elsevier Science Publishers, 1986, pp. 147–163.

[43] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.

[44] H. Schwetman, "CSIM: A C-based, process-oriented simulation language," in *Proc. 1986 Winter Simulation Conference*, 1986, pp. 387–396.

[45] M. Siegle, "Using structured modelling for efficient performance prediction of parallel systems," in *Parallel Computing: Trends and Applications* (G.R. Joubert *et al.*, eds.), North-Holland, 1994, pp. 453–460.

[46] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran, "An approach to scalability of shared memory parallel systems," in *Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, Nashville, May 1994, pp. 171–180.

[47] K. So, A.S. Bolmarcich, F. Darema, and V.A. Norton, "A speedup analyzer for parallel programs," in *Proc. 1987 Int. Conf. Parallel Proc.*, IEEE, Aug. 1987, pp. 653–661.

[48] F. Sötz, "A method for performance prediction of parallel programs," in *Proc. CONPAR 90-VAPP IV (LNCS 457)* (H. Burkhart, ed.), Springer-Verlag, 1990, pp. 98–107.

[49] B. Stramm and F. Berman, "Predicting the performance of large programs on scalable multicomputers," in *Scalable HPC Conference*, Apr. 1992, pp. 22–29.

[50] A. Thomasian and P.F. Bay, "Analytic queueing network models for parallel processing task systems," *IEEE Transactions on Computers*, vol. 35, Dec. 1986, pp. 1045–1054.

[51] H. Wabnig and G. Haring, "Paps - the parallel program performance prediction toolset," in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 794) (G. Haring and G. Kotsis, eds.), Berlin, Springer-Verlag, May 1994.

[52] K-Y. Wang, "Intelligent program optimization and parallelization for parallel computers," Tech. Rep. CSD-TR 91-030, Purdue University, Apr. 1991.

[53] N. Yazici-Pekergin and J-M. Vincent, "Stochastic bounds on execution times of parallel programs," *IEEE Transactions on Software Engineering*, vol. 17, Oct. 1991, pp. 1005–1012.

[54] J. Zahorjan *et al.*, "Balanced job bound analysis of queueing networks," *Communications of the ACM*, vol. 25, Feb. 1982, pp. 134–141.

Analyzing the Behavior and Performance of Parallel Programs

Vikram Adve, Rice University

A task graph provides a natural representation of the inherent parallelism in a parallel program, but previous task-graph-based performance models are only practical to use for programs with simple (fork-join) task graphs. The principal limitations arise because most previous models use non-deterministic task times, which makes it extremely difficult to compute synchronization costs. In this talk, I first briefly describe an analytical and experimental study which provides strong evidence that communication and resource contention delays introduce extremely low variance into process execution times between synchronization points. These results suggest that a deterministic model could be used for performance prediction. In the second part of the talk, I describe a performance prediction model that uses deterministic values to represent mean task times including mean costs of communication and shared-resource contention. The deterministic assumption yields a conceptually simple model that is efficient, accurate, and capable of analyzing programs with large and complex task graphs and sophisticated task scheduling. I use two example programs to illustrate the insight and predictive power provided by the model.

Dynamic Performance Prediction within the ESPRIT PPPE Project

Alistair Dunlop, The University of Southampton, Great Britain

In this talk we present the method and results of performance prediction work done within the ESPRIT PPPE project. The focus of this work has been to provide a low cost, reliable method of predicting performance for message passing Fortran 77 programs. Our approach is based on both static and dynamic program analysis. Minimum and maximum execution times are estimated, and a detailed breakdown of expected functional unit use and data movement within the memory hierarchy is predicted. Results show a substantial improvement in accuracy using this method over more traditional static performance methods.

List of Participants

Vikram Adve

Rice University
CRPC, MS-41
6100 South Main
Houston, Texas 77005
adve@cs.rice.edu

Saman Amarasinghe

Stanford University
Center for Integrated Systems
Stanford, CA 94305
saman@cs.stanford.edu

Jennifer Anderson

Stanford University
Center for Integrated Systems
Stanford, CA 94305
anderson@cs.stanford.edu

Theresa Chatman

Rice University
CRPC
6100 South Main
Houston, Texas 77005
tlc@cs.rice.edu

Beniamino Di Martino

University "Federico II"
Department of Computer Science and Systems
Naples
Italy
dimartin@cps.na.cnr.it
dimartin@par.univie.ac.at

Alistair Dunlop

The University of Southampton
Department of Electronics & Computer Science
Southampton, SO9 5NH
Great Britain
and@ecs.soton.ac.uk

Thomas Fahringer

University of Vienna
Institute for Software Technology and Parallel Systems
Liechtensteinstrasse 22
A-1092 Vienna
Austria
tf@par.univie.ac.at

Guang R. Gao

McGill University
School of Computer Science
Montreal, Quebec, Canada H3A 2A7
gao@cs.mcgill.ca

Jordi Garcia

Universitat Politecnica de Catalunya
c/ Gran Capita s/, Modul D4
Departament d'Arquitectura de Computadors
08034 - Barcelona
Spain
jordig@ac.upc.es

Arjan J.C. van Gemund

Delft University of Technology
Faculty of Electrical Engineering
Laboratory of Computer Architecture and Digital Techniques
P.O. Box 5031
NL-2600 GA Delft
The Netherlands
A.vGemund@et.tudelft.nl

Manish Gupta

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
mgupta@watson.ibm.com

Christoph Kessler

University of Saarland
Computer Science Department
Im Stadtwald
W-6600 Saarbruecken
Germany
kessler@cs.uni-sb.de

Ken Kennedy

Rice University
CRPC, MS-41
6100 South Main
Houston, Texas 77005
ken@cs.rice.edu

Kathy Knobe

Massachusetts Institute of Technology
Artificial Intelligence Lab
545 Technology Square
Boston, MA 02139
kathyk@ai.mit.edu

Uli Kremer

Rice University
CRPC, MS-41
6100 South Main
Houston, Texas 77005
kremer@cs.rice.edu

Krishna Kunchithapadam

University of Wisconsin-Madison
Computer Science Department
1210 W. Dayton Street
Madison, WI 53706
krishna@cs.wisc.edu

Jingke Li

Portland State University
Department of Computer Science
P.O. Box 751
Portland, OR 97207
li@cs.pdx.edu

Amy Lim

Stanford University
Center for Integrated Systems
Stanford, CA 94305
aimee@cs.stanford.edu

John Mellor-Crummy

Rice University
CRPC, MS-41
6100 South Main
Houston, Texas 77005
johnmc@cs.rice.edu

Celso Mendes

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, Il 61801
mendes@cs.uiuc.edu

Daniel Palermo

University of Illinois at Urbana-Champaign
Center for Reliable and High-Performance Computing
1308 West Main Street
Urbana, IL 61801
palermo@crhc.uiuc.edu

Cauligis Raghavendra

Washington State University
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752
raghu@eecs.wsu.edu

Jagannathan Ramanujam

Louisiana State University
Department of Electrical & Computer Engineering
Baton Rouge, LA 70803-5901
jxr@gate.ee.lsu.edu

Dan Reed

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, Il 61801
reed@cs.uiuc.edu

Vivek Sarkar

IBM Software Solutions Division
Application Development Technology Institute
555 Bailey Avenue
San Jose, Ca 95141
vivek_sarkar@vnet.ibm.com

Thomas Sheffler

Research Institute for Advanced Computer Science
Mail Stop T27A-1,
NASA Ames Research Center
Moffett Field, CA 94035-1000
sheffler@riacs.edu

Rob Schreiber

Research Institute for Advanced Computer Science
Mail Stop T27A-1,
NASA Ames Research Center
Moffett Field, CA 94035-1000
schreibr@riacs.edu

Yoshiki Seo

Rice University
CRPC, MS-41
6100 South Main
Houston, Texas 77005
yoshiki@cs.rice.edu

Harikumar Sivaraman

Washington State University
School of Electrical Engineering and Computer Science
Pullman, WA 99164-2752
hsivaram@eecs.wsu.edu

Jaspal Subhlok

Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
jass@cs.cmu.edu

Jodi L. Tims

University of Pittsburgh
Department of Computer Science
Pittsburgh, PA 15260
jt@cs.pitt.edu

Jhy-Chun Wang

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, Il 61801
jcwang@cs.uiuc.edu