Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment

Ulrich Kremer
John Mellor-Crummey
Ken Kennedy
Alan Carle

CRPC-TR93298-S
February, 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Published in *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Christoph W. Kessler (editor), pages 136-152, Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993.

Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment*

Ulrich Kremer†
John Mellor-Crummey
Ken Kennedy
Alan Carle

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251

Abstract

Although distributed-memory message-passing parallel computers are among the most cost-effective high performance machines available, scientists find them extremely difficult to program. Most programmers feel uncomfortable working with a distributed-memory programming model that requires explicit management of local name spaces. To address this problem, researchers have proposed using languages based on a global name space annotated with directives specifying how the data should be mapped onto a distributed memory machine. Using these annotations, a sophisticated compiler can automatically transform a code into a message-passing program suitable for execution on a distributed-memory machine. The Fortran77D and Fortran90D languages support this programming style. Given a Fortran D program, the compiler uses data layout directives to automatically generate a single-program, multiple data (SPMD) node program for a given distributed-memory target machine.

*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 DARPA under contract #DABT63-92-C-0038, and the IBM corporation. 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

To achieve high performance with such programs, programmers must select a good data layout. Current tools provide little or no support for this selection process. This paper describes an automatic data layout strategy being investigated for use in the D programming tools currently under development at Rice University. The proposed technique considers the profitability of dynamic data remapping as it explores a rich search space of reasonable alignment and distribution schemes.

1 Introduction

The goal of the D programming tools project is to develop techniques and tools that aid scientists in the construction of programs in abstract parallel languages such as Fortran D [FHK+90] and High Performance Fortran (HPF) [Hig93]. A short introduction to the Fortran D language is given in the appendix. Developing efficient programs in languages such as HPF or Fortran D can be challenging since understanding the performance implications of small perturbations of the program at the source level requires a deep understanding of the compiler technology upon which the language implementation is based. In particular, understanding the impact of data distributions on the data parallelism that will be realized by the compiler is vitally important for users to be able to write efficient programs.

The primary focus of the D project is on developing program analysis infrastructure to support an intelligent editor that will provide users with detailed information about how effectively an underlying compiler implementation can exploit data parallelism in the program. The D editor will bring together a wide range of program analysis technology including program dependence analysis to identify inherently sequential constraints on the order in which values must be computed, static performance estimation to determine the relative merits of particular data distribution alternatives, dynamic performance information to refine the costs associated with particular design alternatives, and automatic data layout.

A proposed automatic data layout tool for the D system will first determine a set of efficient data decomposition schemes for the entire program. Subsequently, the user will be able to select a region of the input program and the system will respond with a set of potential decomposition schemes and their performance characteristics for the selected region. For each scheme, the tool will provide information about the location and type of the communication operations generated by the compiler. This will enable the user to obtain insight into the characteristics of the program when executed on a distributed memory machine, and the capabilities of the underlying compilation system.

In this paper we focus on automatic data layout techniques for regular problems in the context of an advanced compilation system that allows dynamic data decompositions. We describe an initial analysis framework for reasoning about dynamic data layouts at compile time for programs without subroutine calls. The paper is structured as follows. Section 2 provides a short introduction to the Fortran D compilation system. Section 3 contains examples that motivate the need for dynamic data decomposition in an automatic tool. Section 4 discusses an initial framework to solve the dynamic data decomposition problem. The paper concludes with a discussion of related work and our future plans.

2 Compilation system

The choice of a good data decomposition scheme for a program depends on the compilation system, the target machine and its size, and the problem size [BFKK90, BFKK91, LC90b, GB92, Who92]. Advances in compiler technology make it even more difficult for a programmer to predict the performance resulting from a given data decomposition scheme without compiling and running the program on the specific target system. State-of-the-art compilers perform a variety of intra- and inter-procedural optimizations. The applicability and profitability of these optimizations depend on the specified data decomposition schemes.

Compilation of a Fortran D program involves translating it into a Fortran 77 SPMD node program that contains calls to library primitives for interprocessor communication. A vendor-supplied Fortran 77 node compiler is used to generate an executable that will run on each node of the distributed-memory target machine. A Fortran D compiler may support optimizations that reduce or hide communication overhead, exploit parallelism, or reduce memory requirements. Procedure cloning or inlining may be applied under certain conditions to improve context for optimization [HKT91, HKT92, HHKT91, Tse93]. Node compilers may perform optimizations to exploit the memory hierarchy and instruction-level parallelism available on the target node processor [Car92, Wol92, Bri92].

At present, the principal target of our prototype Fortran D compilation system [Tse93] is the Intel iPSC/860. Eventually, the compilation system will target a variety of distributed-memory multiprocessors such as Intel's iPSC/860 and Paragon, Ncube's Ncube-1 and Ncube-2, and Thinking Machine Corporation's CM-5. Our proposed strategy for automatic data decomposition is intended for use with our state-of-the-art Fortran D compilation system.

3 Dynamic Data Layout: Two Examples

The following program examples illustrate the difficulty of predicting the performance impact of dynamic remapping in the context of an advanced compilation system. For this reason, we believe that an automatic tool is needed to determine when data remapping can be used effectively.

The availability of fast collective communication routines is crucial for the profitability of data realignment and redistribution. In our experiments we used a transpose library routine distributed by Intel in a set of example programs for the iPSC/860.

Two-dimensional Fast Fourier Transform (2D-FFT)

The computation performed by a two-dimensional FFT can be described as a sequence of one-dimensional FFTs (1D-FFTs) along each row of the input array, followed by one-dimensional FFTs along each column. The input array in our example is of type complex. The butterfly version of the 2D-FFT distributes the first dimension of the two-dimensional array. This leads to communication during the computation of the 1D-FFTs along each column. This communication can be avoided if the array is transposed after all 1D-FFTs along each row have been performed. The transpose version uses a row-distribution for the row-wise 1D-FFTs and a column-distribution for the column-wise 1D-FFTs. Both versions were compiled

#procs	problem size	butterfly			transpose			relative
		total	communication only	(% of total)	total	communication only	(% of total)	speed-up
	128 × 128							
2		0.423	0.016	3.8%	0.432	0.019	4.4%	**0.98**
4		0.272	0.061	22.4%	0.217	0.015	6.9%	**1.25**
8		0.207	0.092	44.4%	0.113	0.012	10.6%	**1.83**
16		0.187	0.119	63.6%	0.062	0.011	17.7%	**3.02**
32		0.193	0.147	76.1%	0.042	0.017	40.5%	**4.60**
64		0.160	0.124	77.5%	0.035	0.022	62.8%	**4.57**
	256 × 256							
2		1.731	0.036	2.0%	1.819	0.070	3.8%	**0.95**
4		0.979	0.119	12.1%	0.903	0.050	5.5%	**1.08**
8		0.630	0.181	28.7%	0.459	0.031	6.7%	**1.37**
16		0.485	0.238	49.0%	0.237	0.023	9.7%	**2.05**
32		0.444	0.296	66.6%	0.130	0.023	17.7%	**3.42**
64		0.352	0.250	71.0%	0.081	0.026	32.0%	**4.34**
	512 × 512							
2		7.199	0.057	0.8%	7.822	0.299	3.8%	**0.92**
4		3.812	0.235	6.1%	3.814	0.194	5.0%	**1.00**
8		2.178	0.360	16.5%	1.919	0.108	5.6%	**1.13**
16		1.428	0.474	33.2%	0.969	0.064	6.6%	**1.47**
32		1.127	0.597	53.0%	0.498	0.046	9.2%	**2.26**
64		0.826	0.503	60.9%	0.270	0.040	14.8%	**3.06**
	1024 × 1024							
4		15.640	0.444	2.8%	16.561	0.836	5.0%	**0.94**
8		8.332	0.718	8.6%	8.274	0.432	5.2%	**1.01**
16		4.827	0.939	19.4%	4.156	0.238	5.7%	**1.16**
32		3.324	1.274	48.8%	2.097	0.137	6.5%	**1.59**
64		2.152	1.005	46.7%	1.083	0.090	8.3%	**1.99**
	2048 × 2048							
16		18.323	1.895	10.3%	18.360	0.893	4.9%	**0.99**
32		10.764	2.356	21.9%	9.215	0.487	5.3%	**1.17**
64		6.456	2.007	31.1%	4.687	0.277	5.9%	**1.38**

Table 1: Performance of two versions of 2D-FFT on the iPSC/860

using if77 under -O4 option and executed on the iPSC/860 at Rice (32 processors) and Caltech (64 processors). Both machines have a two-way set associative instruction cache (4Kbytes) and data cache (8Kbytes). The cache lines are 32 bytes long. Table 1 lists execution times in seconds for the butterfly and transpose implementation alternatives over a variety of data sizes and processor configurations. For each implementation alternative, the table lists the total execution time and the fraction of the time spent communicating. The last column lists the relative speed-ups of the transpose version over the butterfly version for different problem sizes and processor configurations. In almost all cases redistribution leads to a better performance as compared to a static row partitioning. The most significant improvements occur for small problems and a high number of processors.

If the compiler is not able to detect the FFT (butterfly) communication pattern, we expect the compiler-generated program for the static row partitioning to run slower than the butterfly version, increasing the benefits of the transpose version even more.

Alternating-Direction-Implicit Integration (ADI)

The sequential code is shown in Figure 1. Each iteration of the DO-loop in line 2 consists of a forward and backward sweep along the rows of arrays `x` and `b`, followed by a forward and backward sweep along the columns. The *pipeline* version of the code uses a static column-wise partitioning of the perfectly aligned arrays `x, a`, and `b`. We specified this data layout using Fortran D language annotations and compiled the program using the current Fortran D compiler prototype. The compiler generated a coarse-grain pipelined loop for the forward and backward sweeps along the rows of arrays `x` and `b`. The sweeps along columns

do not require communication under this data layout, although the row sweeps do. The *transpose* version transposes arrays `x` and `b` between the row and column sweeps, i.e. twice per iteration of the outermost DO-loop (line 2). No communication is needed during each sweep.

The execution times for 10 iterations (MAXITER = 10) for the iPSC/860 is shown in Table 2. The timings are given in seconds. Since the selected problem sizes are powers of two, cache conflicts lead to a significant increase of the total execution time for the transpose version. To alleviate this problem, we added a single column or row to each local segment of the arrays in the node SPMD program. We expect a sophisticated node compiler to perform such an optimization. The performance of the modified node programs are listed under the problem sizes marked with asterisks in Table 2. In contrast to the 2D-FFT example, redistribution leads to a decrease in performance in all cases. The extent of improvement of the pipeline version over the transpose version depends on the ability of the compiler to deal with the cache effects on the target machine.

Discussion

The 2D-FFT example shows that dynamic remapping can result in significant performance improvements over a static data layout scheme. A programmer might have expected a similar performance improvement for the ADI example program. However, due to the coarse grain pipelining optimization performed by the Fortran D compiler dynamic data remapping is not profitable even if we ignore cache conflicts.

4 Towards Dynamic Data Layout

The first step of our proposed strategy for automatic data layout in the presence of dynamic remapping is to partition the program into code segments, called program *phases*. Phases are intended to represent program segments that perform operations on entire data objects. In the absence of procedure calls we operationally define a phase as follows: A phase is a loop nest such that for each induction variable that occurs in a subscript position of an array reference in the loop body the phase contains the surrounding loop that defines the induction variable. A phase is minimal in the sense that it does not include surrounding loops that do not define induction variables occurring in subscript positions. Data remapping is allowed only between phases. Note that the strategies for identifying program phases is a topic of current research.

Our strategy for investigating data layout with dynamic data remapping explores a rich search space of possible alignment and distribution schemes for each phase. Pruning heuristics will have to be developed to restrict the alignment and distribution search spaces to manageable sizes. A first discussion of possible pruning heuristics and the sizes of their resulting search spaces can be found in [KK93].

Here we describe an initial analysis framework suitable for programs without procedure calls that contain no control flow other than loops. We assume that the problem size and the number of processors used is known at compile time. Furthermore, we assume that every

```
1     REAL x(N, N), a(N, N), b(N, N)
2     DO iter = 1, MAXITER
3         // ADI forward & backward sweeps along rows
4         DO j = 2, N
5             DO i = 1, N
6                 x(i, j) = x(i, j) − x(i, j−1) ∗ a(i, j) / b(i, j−1)
7                 b(i, j) = b(i, j) − a(i, j) ∗ a(i, j) / b(i, j−1)
8             ENDDO
9         ENDDO
10        DO i = 1, N
11            x(i, N) = x(i, N) / b(i, N)
12        ENDDO
13        DO j = N−1, 1, −1
14            DO i = 1, N
15                x(i, j) = ( x(i, j) − a(i, j+1) ∗ x(i, j+1) ) / b(i, j)
16            ENDDO
17        ENDDO
18        // ADI forward & backward sweeps along columns
19        DO j = 1, N
20            DO i = 2, N
21                x(i, j) = x(i, j) − x(i−1, j) ∗ a(i, j) / b(i−1, j)
22                b(i, j) = b(i, j) − a(i, j) ∗ a(i, j) / b(i−1, j)
23            ENDDO
24        ENDDO
25        DO j = 1, N
26            x(N, j) = x(N, j) / b(N, j)
27        ENDDO
28        DO j = 1, N
29            DO i = N−1, 1, −1
30                x(i, j) = ( x(i, j) − a(i+1, j) ∗ x(i+1, j) ) / b(i, j)
31            ENDDO
32        ENDDO
33    ENDDO
```

Figure 1: Sequential ADI code, `REAL`

#procs	problem size	pipeline			transpose			relative
		total	communication only	(% of total)	total	communication only	(% of total)	speed-up
	128 × 128							
2		2.305	0.021	0.9%	3.894	0.371	9.5%	**0.59**
4		1.265	0.053	4.2%	1.547	0.298	19.3%	**0.82**
8		0.720	0.077	10.7%	1.371	0.246	17.9%	**0.52**
16		0.485	0.103	21.2%	0.617	0.280	45.4%	**0.78**
32		0.404	0.141	34.9%	1.137	0.444	39.0%	**0.35**
64		0.431	0.235	54.5%	1.130	0.821	72.6%	**0.38**
	128 × 128*							
2		2.283	0.020	0.9%	2.486	0.365	14.7%	**0.92**
4		1.281	0.053	4.1%	1.381	0.308	22.3%	**0.93**
8		0.715	0.080	11.1%	0.835	0.255	30.5%	**0.85**
16		0.505	0.116	23.0%	0.614	0.348	56.7%	**0.82**
32		0.402	0.143	35.6%	0.596	0.454	76.2%	**0.67**
64		0.430	0.236	54.9%	0.921	0.845	91.7%	**0.47**
	256 × 256							
2		9.142	0.041	0.4%	21.585	1.351	6.2%	**0.42**
4		4.781	0.106	2.2%	10.261	1.009	9.8%	**0.46**
8		2.598	0.162	6.2%	4.250	0.677	15.9%	**0.61**
16		1.531	0.180	11.7%	3.192	0.532	16.6%	**0.48**
32		1.064	0.215	20.2%	2.050	0.607	29.6%	**0.52**
64		0.838	0.307	36.6%	3.046	0.921	30.2%	**0.27**
	256 × 256*							
2		9.045	0.042	0.5%	10.116	1.368	13.5%	**0.89**
4		4.758	0.106	2.2%	5.296	1.009	19.0%	**0.90**
8		2.566	0.167	6.5%	2.844	0.678	23.8%	**0.90**
16		1.566	0.204	13.0%	1.709	0.596	34.9%	**0.92**
32		0.986	0.220	22.3%	1.235	0.623	50.4%	**0.80**
64		0.828	0.320	38.6%	1.255	0.958	76.3%	**0.66**
	512 × 512							
2		39.553	0.084	0.2%	161.896	5.504	3.4%	**0.24**
4		20.270	0.209	1.0%	81.868	3.717	4.5%	**0.25**
8		10.612	0.308	2.9%	39.072	2.289	5.8%	**0.27**
16		5.780	0.352	6.0%	10.980	1.439	13.1%	**0.53**
32		3.406	0.379	11.1%	6.576	1.110	16.9%	**0.52**
64		2.289	0.468	20.4%	5.704	1.257	22.0%	**0.40**
	512 × 512*							
2		35.913	0.083	0.2%	144.500	5.561	3.8%	**0.25**
4		18.434	0.207	1.1%	21.365	3.768	17.6%	**0.86**
8		9.573	0.313	3.3%	10.795	2.270	21.0%	**0.89**
16		5.302	0.372	7.0%	5.799	1.497	25.8%	**0.91**
32		3.060	0.376	12.2%	3.329	1.132	34.0%	**0.92**
64		2.083	0.467	22.4%	2.402	1.296	54.0%	**0.87**
	1024 × 1024							
2		168.055	0.175	0.1%	949.241	27.171	2.9%	**0.18**
4		85.106	0.411	0.5%	388.358	15.016	3.9%	**0.22**
8		43.605	0.602	1.4%	237.368	8.449	3.5%	**0.18**
16		22.860	0.678	3.0%	98.511	4.854	4.9%	**0.23**
32		12.509	0.686	5.5%	52.708	2.969	5.6%	**0.24**
64		7.351	0.788	10.7%	13.927	2.283	16.4%	**0.53**
	1024 × 1024*							
2		147.682	0.177	0.1%	752.135	27.400	3.6%	**0.20**
4		73.467	0.410	0.5%	352.282	15.162	4.3%	**0.21**
8		37.484	0.608	1.6%	43.869	8.534	19.4%	**0.85**
16		19.860	0.714	3.6%	22.066	4.866	22.0%	**0.90**
32		10.820	0.683	6.3%	11.629	2.980	25.6%	**0.93**
64		6.466	0.786	12.1%	6.727	2.325	34.6%	**0.96**
	2048 × 2048							
4		337.115	0.909	0.3%	*memory*	*memory*		
8		170.599	1.219	0.7%	815.495	34.042	4.2%	**0.21**
16		87.407	1.365	1.6%	602.598	17.979	3.0%	**0.14**
32		45.911	1.353	2.9%	193.612	10.004	5.2%	**0.24**
64		25.098	1.452	5.8%	131.027	6.052	4.6%	**0.19**
	2048 × 2048*							
4		*memory*	*memory*		*memory*	*memory*		
8		146.694	1.241	0.8%	671.508	34.311	5.1%	**0.22**
16		75.563	1.414	1.9%	89.174	18.161	20.4%	**0.85**
32		39.464	1.353	3.4%	44.293	9.913	22.4%	**0.89**
64		21.691	1.450	6.7%	23.345	6.078	26.0%	**0.93**

Table 2: Performance of pipeline and transpose versions of ADI on the iPSC/860

alignment and distribution scheme specifies the data layout of all arrays in the program that may be partitioned and mapped onto different local memories of the machine.

A data layout for a program is determined in three steps. First, alignment analysis builds a search space of reasonable alignment schemes for each phase. Then, distribution analysis uses the alignment search spaces to build decomposition search spaces of reasonable alignments and distributions for each phase. Finally, a decomposition scheme for each phase is selected, resulting in a data layout for the entire program. Our three step approach to automatic data layout is described in more detail in the following sections. A summary of the basic algorithm is shown in Figure 4.

4.1 Alignment Analysis

Alignment analysis is used to prune the search space of all possible array alignments by selecting only those alignments that minimize data movement. Alignment analysis is largely machine-independent; it is performed by analyzing the array access patterns of computations in each individual program phase and across the entire program. All alignment schemes are specified relative to the *alignment space* of the program. The alignment space of a program is unique. It is determined by the maximal dimensionalities and maximal dimensional extents of the arrays in the program.

We intend to build on the inter-dimensional and intra-dimensional alignment techniques of Li and Chen [LC90a], Knobe *et al.* [KLS90], and Chatterjee, Gilbert, Schreiber, and Teng [CGST93]. In contrast to previous work, we will not limit ourselves to a single alignment as the result of the alignment analysis. Rather than eliminating one candidate alignment in the presence of an alignment conflict, both schemes may be added to the alignment search space [KK93]. The candidate alignments computed for each phase, will serve as input to distribution analysis.

4.2 Distribution Analysis

Distribution analysis will consider a rich set of distribution schemes for each of the alignment schemes determined in the alignment analysis. Each dimension of a decomposition can have a block, cyclic, or block-cyclic distribution [FHK+90]. Block-cyclic distributions can have different block sizes. In addition, distributions with varying numbers of processors in each of the distributed dimensions of a decomposition are part of the distribution search space. We are currently developing strategies to prune the search space by eliminating candidate distributions that are poorly matched to the program being analyzed. The result of distribution analysis will be a set of candidate decomposition schemes for each single phase in the program. For each phase, a static performance estimator will be invoked to predict the performance of each candidate scheme. The resulting performance estimates will be recorded with each decomposition scheme. A performance estimator suitable for our needs is described in detail elsewhere [BFKK91, HKK+91].

4.3 Inter-Phase Decomposition Analysis

After computing a set of data decomposition schemes and estimates of their performance for each phase, the automatic data partitioner must solve the *inter-phase* decomposition problem to choose the best data decomposition for each phase. It must consider array remapping between computational phases to reduce communication costs within the computational phases or to better exploit available parallelism. Inter-phase analysis is performed on a *phase control flow graph*. A phase control flow graph is a control flow graph [ASU86] in which all nodes in a phase have been collapsed into a single node. Inter-phase analysis first detects the strongly connected components of the phase control flow graph in a hierarchical fashion using, for example, Tarjan intervals [Tar74]. For each innermost loop, the inter-phase decomposition selection problem is formulated as a single-source shortest path problem over the acyclic *decomposition graph* associated with the loop body. The decomposition graph is similar to the phase control flow graph except that each phase node is replaced by the candidate set of decompositions for that phase, and for each cycle in the graph a shadow copy of the first phase in the cycle is added after the last phase in the cycle. Each decomposition node is labeled with its estimated overall cost for the phase. The overall cost is determined by computational costs and costs due to synchronization and communication inside the phase. Shadow decomposition nodes are assigned a weight of zero. The flow edges in the phase control flow graph are replaced by the set of all possible edges between decomposition nodes of adjacent phases. The edges are labeled with the realignment and redistribution costs to map between the source and sink decomposition schemes. Edge weights will be determined based on the training set approach [BFKK91]. An example phase control flow graph with a single loop and the decomposition graph associated with the loop body is shown in Figure 2. For clarity the weights of nodes and edges have been omitted.

The root nodes in the decomposition graph represent entry/exit decomposition schemes for the loop. For each root node a single-source shortest path problem is solved. The length of the shortest path between the root node and its shadow copy multiplied by the number of iterations of the loop gives the cost estimate of the loop for the associated entry/exit decomposition scheme[1]. After determining the costs of each decomposition scheme for an innermost loop, the loop is collapsed into a single *loop summary node* in the phase control flow graph. The algorithm records with the loop summary phase the costs for each entry/exit decomposition scheme and their associated shortest paths. Subsequently, the process of detecting innermost loops, solving the single-source shortest paths problem, and collapsing the loops into single nodes continues until the phase control flow graph is acyclic. The final step of the merging algorithm consists of solving a single single-source shortest paths problem on a decomposition graph for the entire program with added entry and exit decomposition nodes. The added nodes and their adjacent edges have zero weight. For our example program in Figure 2 the final step is illustrated in Figure 3.

Inter-phase analysis selects the decomposition schemes that lie on the shortest path from the added entry decomposition node to the added exit node. Decomposition nodes that represent a loop summary phase are expanded into their associated shortest paths. Following

[1]The length of a path includes the weights on the nodes as well as the weights of the edges along the path.

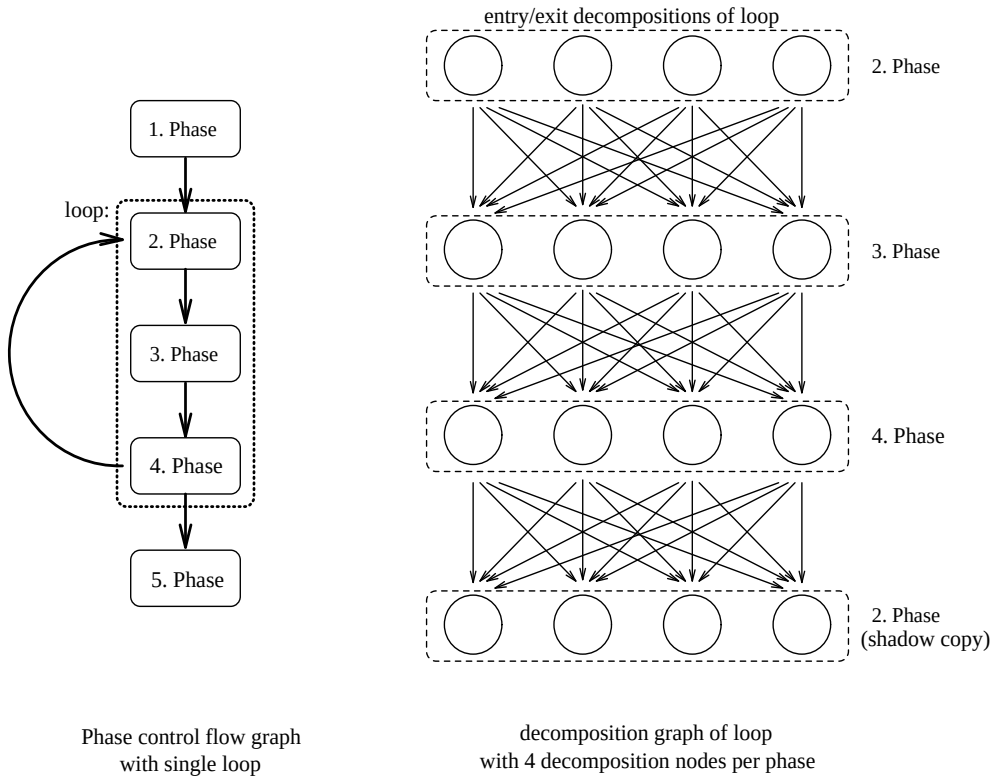

Figure 2: Inter-phase decomposition problem with realignment and redistribution

the selected shortest path, dynamic remapping is required if the decomposition at the source of an edge is different from the decomposition at the sink.

A solution to the single-source shortest paths problem in a directed acyclic graph is given in [CLR90]. Let k denote the maximal number of decomposition schemes for each phase and p the number of phases. The resulting time complexity is $\mathcal{O}(pk^3)$. The identification of innermost loops takes time proportional to the number of edges in the phase control flow graph. For our class of control flow graphs $\mathcal{O}(edges) = \mathcal{O}(nodes)$ holds, resulting in $\mathcal{O}(p)$ time for Tarjan's algorithm [Tar74]. Therefore, the entire algorithm for merging decomposition schemes across phases has time complexity $\mathcal{O}(pk^3)$.

It is important to note that the presented solution to the merging problem assumes that each decomposition scheme specifies the data layout of every array in the program that may be partitioned across the machine. If we relax this restriction, for instance by allowing each decomposition scheme for a phase to only specify the layout of arrays actually referenced in the phase, the dynamic data layout problem becomes NP-complete [Kre93b]. A study of real programs will show when the restriction has to be relaxed in order to limit the number of candidate decomposition schemes for each phase. We are currently working on heuristics that will generate efficient, but possibly suboptimal data layouts under the relaxed restriction.

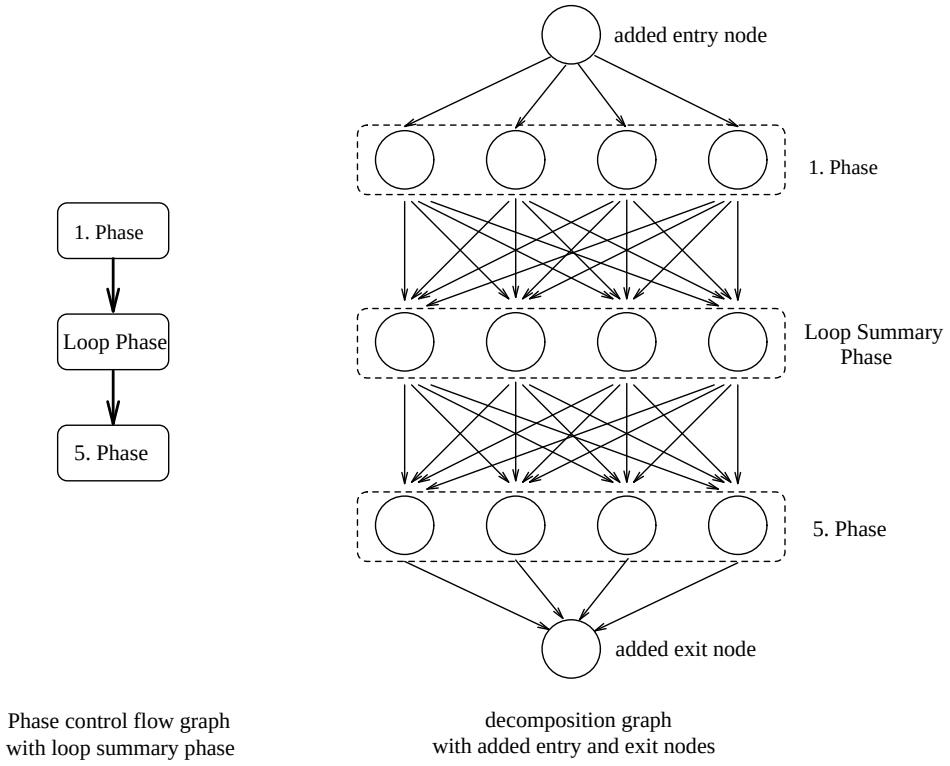

Figure 3: Inter-phase decomposition problem with realignment and redistribution (cont.)

5 Related Work

The problem of finding an efficient data layout for a distributed-memory multiprocessor has been addressed by many researchers [LC90a, LC91, KLS90, KN90, KLD92, Gup92, Who91, CGST93, AL93, CHZ91, RS89, Ram90, HA90, SS90, Sus91, IFKF90]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data decompositions, the compilation system, and the target distributed-memory machine. A more detailed discussion of some of the related work can be found in [Kre93a]. Our work is one of the first to provide a framework for automatic data layout that considers dynamic remapping. However, many researchers have recognized the need for dynamic remapping and are planning to develop solutions.

Knobe, Lukas, and Dally [KLD92], and Chatterjee, Gilbert, Schreiber, and Teng [CGST93] address the problem of dynamic alignment in a framework particularly suitable for SIMD machines. More recently, Anderson and Lam [AL93] have proposed techniques for automatic data layout for distributed and shared address space machines. Their approach considers dynamic remapping.

Algorithm DECOMP

Input: program without procedure calls; problem sizes and number of processors to be used.
Output: data layout for data objects referenced in the input program

Determine program phases of input program; build phase control flow graph.

Perform alignment and distribution analysis for input program; each resulting decompositition scheme specifies data layout of all arrays that may be partitioned.

while phase control flow graph contains a loop *do*
    Identify innermost loop (e.g. using Tarjan intervals).
    Build decomposition graph for innermost loop body.
    Solve single-source shortest paths problem on decomposition graph.
    Replace loop by its summary phase in the phase control flow graph; record
    cost of entry/exit decomposition schemes together with their shortest paths.
endwhile

Build decomposition graph for collapsed phase control flow graph.
Add entry and exit decomposition nodes.
Solve single-source shortest paths problem on decomposition graph.

Determine data layout for entire program by traversing the lowest cost
shortest path from entry node to exit node, expanding loop summary
phases by their associated shortest paths.

Figure 4: Automatic Data Layout Algorithm

6 Summary and Future Work

This paper presents an initial framework for automatic data decomposition that allows dynamic remapping between program phases. Our proposed strategy explores a rich search space of alignment and distribution schemes for each program phase. The costs of decomposition schemes for program phases and the data remapping costs between phases will be computed by a static performance estimator. The data layout for the entire program is determined based on solutions to single-source shortest paths problems. These solutions require that each decomposition scheme specifies the layout of all arrays in the program that may be partitioned. For the proposed approach to be feasible, we will need to develop algorithms that will prune the alignment and distribution search spaces.

Relaxing the requirement for decompositions, i.e. allowing decomposition schemes to only specify the layout of a subset of the arrays in the program, makes the inter-phase de-

composition problem NP-complete. We will need to explore whether the proposed framework will be practical for real programs or whether heuristics will have to be used to solve the inter-phase decomposition problem. The framework will be extended to handle intra-phase and inter-phase control flow, and to allow programs that consist of a collection of procedures.

We propose using data layout analysis as the basis for a tool that will enable a user to select a region of the input program and have the tool respond with the set of decomposition schemes in its search space and their performance characteristics for the selected region. Such a tool will help the user to understand the impact of data layout schemes on the performance of the program executed on a target distributed-memory machine, and the characteristics of the underlying compilation system. The exact design and functionality of the interface between the user and the automatic data layout tool is currently under development.

References

[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.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[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.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[Car92] S. Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, September 1992.

[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.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[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.

[GB92] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, April 1992.

[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.

[HHKT91] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Technical Report TR91-169, Dept. of Computer Science, Rice University, November 1991.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993. To appear in *Scientific Programming*, vol. 2, no. 1.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KK93] K. Kennedy and U. Kremer. Initial framework for automatic data layout in Fortran D: A short update on a case study. Technical Report CRPC-TR93-324-S, Center for Research on Parallel Computation, Rice University, July 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, July 1992.

[KLS90] 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.

[KM91] 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.

[KN90] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[Kre93a] U. Kremer. Automatic data layout for distributed-memory machines. Technical Report CRPC-TR93-299-S, Center for Research on Parallel Computation, Rice University, February 1993. (thesis proposal).

[Kre93b] 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.

[LC90a] 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.

[LC90b] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Dept. of Computer Science, Yale University, New Haven, CT, May 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.

[Ram90] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1990.

[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.

[SS90] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Sus91] A. Sussman. *Model-Driven Mapping onto Distributed Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, September 1991.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[TMC89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 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.

[Who92] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[Wol92] M.E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, August 1992.

A Fortran D Language

The task of distributing data across processors can be approached by considering the two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors of the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D is a version of Fortran that provides data decomposition specifications for these two levels of parallelism using `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The `DECOMPOSITION` statement declares the name, dimensionality, and size of a decomposition.

The `ALIGN` statement maps arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders `I, J, K, ...` in the subscript expressions of both the array and decomposition. In the example below,

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J-2,I+3)
```

A is declared to be a two dimensional decomposition of size $N \times N$. Array X is then aligned with respect to A with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are `BLOCK`, `CYCLIC`, and `BLOCK_CYCLIC`. The symbol ":" marks dimensions that

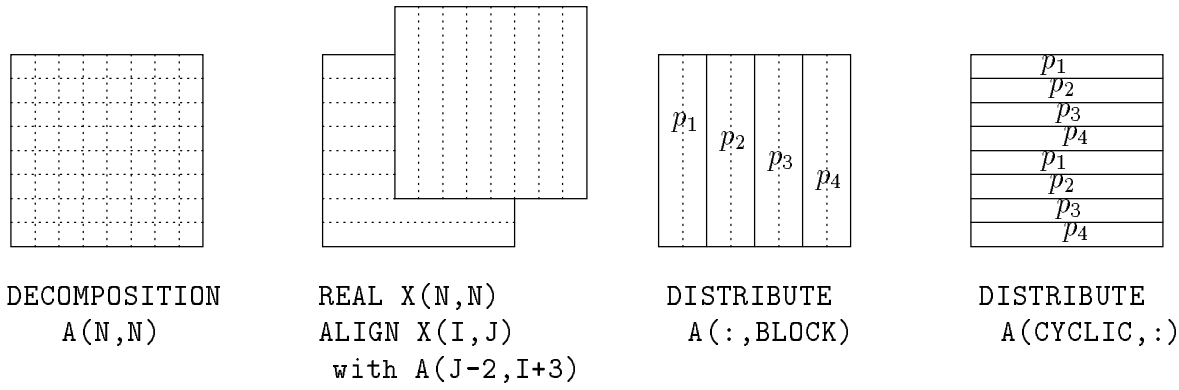

Figure 5: Fortran D Data Decomposition Specifications

are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. In the following example,

```
DECOMPOSITION A(N,N), B(N,N)
DISTRIBUTE A(:, BLOCK)
DISTRIBUTE B(CYCLIC,:)
```

distributing decomposition A by `(:,BLOCK)` results in a column partition of arrays aligned with A. Distributing B by `(CYCLIC,:)` partitions the rows of B in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 5.

We should note that the goal in designing Fortran D is not to support the most general data decompositions possible. Instead, the intent is to provide decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Fortran D is a language with semantics very similar to sequential Fortran. As a result, it should be easy to use by computational scientists. In addition, we believe that the two-phase strategy for specifying data decomposition is natural and conducive to writing modular and portable code. Fortran D bears similarities to CM Fortran [TMC89], Kali [KM91], and Vienna Fortran [CMZ92]. The complete language is described in detail elsewhere [FHK+90].