# Compiler Support for Machine Independent Parallelization of Irregular Problems

*Reinhard von Hanxleden*

**CRPC-TR92301-S**
**November 1992**

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

# Compiler Support for Machine Independent Parallelization of Irregular Problems

Reinhard von Hanxleden

Thesis Proposal

November 1992

*Department of Computer Science*
*Rice University*
*Houston, TX 77251*
`reinhard@rice.edu`

**Abstract**

The Fortran D group at Rice University aims at providing a machine independent data parallel programming style, in which the applications programmer uses a dialect of sequential Fortran and high level distribution annotations. Extracting parallelism from these applications typically is straightforward, but making efficient use of this parallelism for irregular applications, such as molecular dynamics or unstructured meshes, is a challenge due to the limited compile-time knowledge about data access patterns.

It is my thesis that the spatial locality of the underlying problems can be used as a basis of compiler support for parallelizing such applications. *Value-based decompositions* are an extension of Fortran D to express the spatial locality of an application and to assist the compiler in computing a distribution with both a balanced computational workload and high data access locality. A *communication data flow framework* detects opportunities to combine messages, move them into less frequently executed code regions, or even eliminate them. *Loop flattening* is a code transformation to overcome SIMD specific control flow limitations when executing nested loops with varying inner loop bounds, which are typical for irregular problems.

## 1 Introduction

The Fortran D group at Rice University aims at providing a "machine independent parallel programming style," in which the applications programmer uses a dialect of sequential Fortran and annotates it with high level distribution information [FHK+90]. From this annotated program, Fortran D compilers will generate codes in different native Fortran dialects for different parallel architectures. The target architectures include both shared and distributed memory architectures and both MIMD and SIMD machines. The overall goal for the code generated by the Fortran D compiler is to have a performance which is, for most applications, relatively close to the performance of hand written native code.

A prototype FORTRAN D compiler targeting the iPSC/860 has been under development. This compiler has had considerable success with regular problems [HHKT91, HKK+91, HKT91, HKT92b]. The goal of my research is to extend the expressiveness and effectiveness of the FORTRAN D constructs like `DECOMPOSITION`, `DISTRIBUTE`, `ALIGN`, and `FORALL` into the domain of irregular problems.

## 1.1 Irregular Problems

Depending upon one's point of view, there are different criteria for when a problem or an application is irregular.

- A physicist might classify a computational problem by the degree of geometric simplicity and the variance of density of the underlying physical problem. An example of a regular problem under this metric is the calculation of a simple wave equation for a rectangular problem domain, whereas mapping out the gravity potential for an expanding galaxy is certainly of irregular nature.
- An applied mathematician who sees the mathematical description of a problem can consider the sparsity of the data describing a particular instance of the problem. Typical examples are finite difference methods, which have a dense (regular) description, *vs.* finite element methods, whose description is sparse due to the varying element sizes.
- A computer scientist is typically interested in the complexity of the data structures and access mechanisms needed to efficiently solve a problem; "efficiently" because even irregular problems can usually be captured by very simple data structures, but not without wasting memory and/or processing power. Simple arrays, accessed directly, are typical for regular problems, whereas irregular applications may employ arrays with indirection vectors, pointers, linked lists, or quad trees, for example.

We consider a problem to be irregular if its data access patterns are hard to analyze at compile time; *i.e.*, there is no obvious, simple parallelization that gives good speedups and makes efficient use of processing power and storage capacities. This is usually associated with the need for sophisticated data structures, for example quad trees for solving $N$-body problems. These data structures are more difficult to distribute among distributed memory processors than, for example, simple arrays accessed in a regular manner. However, they enable us to use faster algorithms; in the $N$-body example, we can replace the naïve $\mathcal{O}(N^2)$ algorithm with hierarchical tree methods, having an $\mathcal{O}(N \log N)$ [BH86] or even an $\mathcal{O}(N)$ time bound [GR87].

With increasing processor power opening the door to solving scientific problems that were previously impractical to solve ("Grand Challenges"), the relative importance of the already widespread irregular applications is expected to increase even further. Examples for areas of high interest are molecular dynamics, galaxy simulation, gene decoding, climate modeling, and computational fluid dynamics. Typical difficulties when parallelizing irregular problems are:

**Difficulty 1** *Bad load balance, for example when modeling a rapidly changing physical system.*

**Difficulty 2** *Lack of compile time knowledge about where and which data have to be communicated, for example in Monte-Carlo processes.*

**Difficulty 3** *Limited locality, for example when computing long range interactions between particles.*

**Difficulty 4** *Large communication requirements, for example when simulating many timesteps of a relatively small, but dynamic system.*

The thesis of this dissertation is the following:

> *It is feasible and profitable to provide compiler support for the parallelization of scientific applications of irregular nature to* directly *exploit the spatial locality of the underlying problem. An important component of this compiler support is the concept of* value-based decompositions *that are derived from snapshots of the spatial configuration of the application. In combination with the alignment mechanism provided in data parallel languages such as* FORTRAN D*, value-based decompositions are a practical and convenient handle for expressing both spatial locality and data interdependence.*

Here, as in the rest of this proposal, the term *spatial locality* refers to the physical locality in the problem domain of the application (as opposed to locality of reference in an array, for example). Furthermore, the term *data interdependence* is used in a high-level sense, like "the force between two particles depends on their distance" (as opposed to the field of dependence analysis which derives statement ordering constraints based on definitions and uses of the same variables).

The goal for this dissertation is to prove this thesis and to extend the support for irregular problems within the FORTRAN D framework. Some points of particular interest are:

- Scalability towards large-scale parallelism ( $\geq$ 1K processors),
- Fundamental and practical differences between MIMD and SIMD architectures with respect to irregular problems, and
- The impact of high message latencies.

The rest of this proposal is organized as follows. Section 2 gives some background about parallelizing irregular problems and describes related work. This section is organized into a hierarchy of levels reaching from the high level algorithm formulation of the application down to the target machine architecture. Section 3 gives an overview about the actual research plan, the contributions of this thesis, and the validation approach. Section 4 concludes with a very brief summary and a description of some possible spin-offs this work will make possible. Appendices A, B, and C contain further details of the projects outlined in Section 3.

## 2 Background and Related Work

We can classify the steps towards the efficient parallel implementation of an irregular problem by the level at which they are taken: the algorithm development, the program text, supportive tools, the compiler, the operating system, or the underlying hardware.

While this thesis has a strong focus on the compiler level, this section lays the ground for putting its contributions in perspective by examining related work also at higher and lower levels and in the domain of regular applications.

## 2.1 The Choice of the Algorithm

The algorithm embodies many high level decisions that affect a parallel implementation. Obviously it is the key towards good efficiency, and it is often worth redesigning a preexisting sequential algorithm before taking it as a basis for a parallel implementation. An typical example can be found in the WaTor population simulation [FJL+88]. In WaTor, members of different species breed, move around within a quantized space, and die. To avoid overpopulation, one rule is that whenever two animals move to the same location, one of them has to retreat and try to go elsewhere. In a sequential implementation, the overhead associated with this *rollback* is usually insignificant. In a parallel implementation, however, a rollback across processor boundaries can be very expensive. Due to potential race conditions, it also becomes difficult to assure determinism, even for a fixed number of processors and a fixed problem decomposition. After identifying such a problem, one should reexamine the algorithm, which was originally formulated for sequential computers. In the WaTor case, it turns out that for enforcing the concept of finite space and avoiding overpopulation, there are alternatives to the original rollback rule [HS92]. These alternatives are not only more efficient on parallel machines by avoiding costly rollback operations, they also enable deterministic program execution even for different problem decompositions and for varying numbers of processors.

In practice, redesigning algorithms for parallel implementations (which are typically more complicated than the simplistic WaTor application) is often hampered by the algorithm designer having a solid expertise in either the underlying application or the performance characteristics of parallel target machines, but usually not in both. However, with the field of parallel programming slowly maturing, parallel algorithms have been formulated for many important application domains [Fox91]. While the results of this thesis may have some implications for the design of good parallel algorithms for irregular problems as well, they are not its main concern.

## 2.2 The Source Program

The program is often written by the same person as the algorithm, but it is responsible for a different set of decisions. Very important is the choice of the right data structures, and for irregular applications in particular many tradeoffs have to be made. Some of these tradeoffs are listed here.

**Simplicity:** Critical in the development stage, but also for later maintenance.

For example, one might argue that the long lasting success of molecular dynamics packages like Gromos [GB88] or Charmm [BBO+83] is at least partially due to the simplicity of their main data structures. These programs use standard arrays with indirect accesses, as opposed to complicated pointer structures with data allocation/deallocation or other data structures that capture spatial locality. Although Fortran is the underlying language in both cases, it is only part of the reason for this simplicity, since one might simulate data structures beyond arrays here as well. At least equally important seems to be the background of the scientists who made the first implementations, whose ugly but straightforward programming style enabled ongoing modifications and vast extensions of these packages by their colleagues.

**Simplicity:** Again, but now from the compiler's point of view. Here not only the data structures themselves are important, but also the way they are accessed.

Coming back to the molecular dynamics example, it might be (relatively) easy for a human to distinguish between code regions computing nonbonded forces for solute atoms and corresponding code regions for solvent atoms and see that they do not interfere with each other. A compiler, however, sees that there are code regions manipulating the same arrays and might therefore overlook possible optimizations.

**Efficiency and Locality:** This is often a space/time tradeoff. For example, it takes fewer instructions to access an array element directly than to do so indirectly, but direct access often implies large storage needs (see Appendix A). For parallel applications, *coarse locality*, *i.e.*, locality on a per-processor view, is crucial for keeping communication costs down. However, data structures that enable organization of data according to the spatial proximity of the entities they are related to (stars, atoms, ...) have some overhead which has to be traded off against communication efficiency.

Equally important is the *fine-grain locality*, which is critical for good performance on the memory hierarchy below off-processor access. If we let $c_{primary}$, $c_{secondary}$, and $c_{comm}$ be the cost of accessing a piece of data from primary memory (registers, cache), secondary memory (main memory), and off-processor, respectively, it seems that $f_{local} = c_{secondary}/c_{primary}$ is approaching $f_{global} = c_{comm}/c_{secondary}$ on newer architectures. Furthermore, as soon as we have a reasonable coarse locality, the factor $f_{local}$ comes into play much more often than $f_{global}$. (Unfortunately it seems that this fine locality is often much less carefully considered than coarse locality. The reason is probably that it easier to picture the cost of fetching data from another processor, which might be several feet away, than the concept of finite cache lines, associativity, and so on.) Here the use of indirection obviously has a high impact again. For example, the poor performance of many molecular dynamics codes on vector machines such as the Cray is basically a consequence of low fine-grain locality. In this case, the factors that prevent a code from vectorization are similar to those that cause frequent cache misses [SM90].

## 2.3 Tools

Tools try to resolve the tradeoffs between simplicity and efficiency. They can assist in load balancing and in communication, both of which can be particularly tedious and error prone when trying to parallelize an irregular problem efficiently [HS91b].

The tools for parallelizing irregular problems can roughly be divided into two groups. The first group of tools provides an easier grip on the physical properties of the problem, *i.e.*, it takes advantage of *spatial* locality. The second group comes into play after the data structures have been laid out; these tools try to free the user from dealing with the access properties of the parallel program; *i.e.*, they examine *data* locality.

### 2.3.1 Tools based on spatial decomposition

**The Generic Multiprocessor** The *Generic Multiprocessor* (GenMP) [Bad87a, Bad87b, Bad91], aims at providing a machine independent programming environment for a certain class of problems, namely scientific calculations that are spatially localized on a mesh. GenMP is a layer of software that can be thought of as a virtual machine that operates on a $d$-dimensional *work mesh* through a sequence of states. With the aid of application dependent routines written by the user, it repartitions the work mesh across processors to achieve a balanced work load and performs the necessary communication. Good results have been achieved with the implementation of the vorticity-stream function formulation of Euler's equation for incompressible flow in two dimensions in an infinite domain. The tests were performed on a 32 processor distributed memory iPSC hypercube and a 4 processor shared memory Cray X-MP vector machine. The limitation of this approach lies in its specificity towards mesh-based, localized applications, which we try to overcome by using general value-based decompositions as introduced in Section 3.2.

**Lattice Parallelism** *Lattice Parallelism* (LPar) [Bad92] is an SPMD programming model that supports coarse-grained parallelism based on the Fidil language [HC88] and the owner computes rule. It is intended for non-uniform computations that involve partial differential equations and have local structure. It explicitly excludes unstructured calculations such as sparse matrix linear algebra and finite element problems. Its main data type is the *Map*, whose elements are indexed by tuples just like array elements. However, the index set of a map, the *Domain*, is not necessarily rectangular, but can be arbitrarily sparse instead. Furthermore, a map declaration itself does not reserve any storage; this has to be allocated explicitly or by an initialization assignment. Maps are flexible; their domain can change at run time, and several domain constructors (for rectangular domains, also with arbitrary starting and ending points and strides) and operators (union, intersection) are available.

Parallelism is expressed by mapping a logical processing Domain onto a spatial processing Domain. LPar supports load balancing and *ghost regions*, in which each processor stores data within a certain proximity to its own data, similarly to overlap regions [Ger90]. It is currently implemented in C++ for the iPSC/860. LPar treats parallelism at a very high level, it manipulates the structure of the data, rather than the data itself. It enables very elegant formulations of a limited class of problems and can be seen as a potential user of an implementation of the value-based decompositions proposed in Section 3.2. Citing Baden [Bad92]:

> It is not an implementation-level system, and relies on application libraries or other run time systems to handle data partitioning or to handle machine-level optimizations, that could be provided for example by Dino or by Fortran D.

### 2.3.2 Tools based on access patterns

**The inspector-executor paradigm** An important concept for the tools that are based on access patterns is is the *inspector-executor* paradigm [KMV90, MSS+88, KMSB90, WSBH91], which was developed to support message blocking even in the presence of indirection arrays. A loop that contains indirect accesses to a distributed array is processed in four steps:

1. The *inspector* runs through the loop and only records which array elements are accessed, without doing the actual computation. *Communication schedules* are computed that satisfy the communication requirements induced by these access patterns.
2. A *gather* operation fetches all referenced off-processor data from their owners and buffers them locally.
3. The *executor* runs through the loop and performs the computation.
4. A *scatter* writes all off-processor data that have been defined in the loop back to their owners.

**Parti** The PARTI primitives (Parallel Automated Runtime Toolkit at ICASE) are a set of high level communication routines that provide convenient access to off-processor elements of arrays that are accessed (and distributed) irregularly [BS90, SBW90]. PARTI is first to propose and implement user-defined irregular distributions [MSS$^+$88] and a hashed cache for nonlocal values [MSMB90]. They build on the inspector-executor paradigm described above; they

1. Coordinate interprocessor data movement,
2. Manage the storage of and access to copies of off-processor data, and
3. Support a shared name space by building a distributed translation table [SCMB90] to store the local address and processor number for each distributed array element.

Communicating the right data at the right time and place is a difficult, yet crucial task for parallelizing irregular problems. The PARTI primitives are valuable tools for the first part of the problem, namely for determining where to find which data and for efficient data exchange. The dataflow framework presented as part of this proposal in Section 3.3 is designed for attacking the second part of the problem, namely enabling the compiler to make good use of these primitives without further advice by the user.

**The Communication Compiler** The *Communication Compiler* [Dah90] is a software facility for scheduling general communications on the Connection Machine. It employs simulated annealing to find a data mapping with as low communication requirements as possible. It uses a recursive routing algorithm to determine an actual communication schedule. For fixed communication patterns, the cost of generating this schedule can be amortized by reusing, for example, over many time steps of a simulation. Its generality makes it very well applicable towards irregular communication structures. However, the communication patterns have to be known before using the Communication Compiler.

## 2.4 The Compiler

The compiler support level is the focus of this thesis. A principal reason for developing powerful compilers is to shift responsibilities for tedious low level details away from the programmer. This is typically associated with a tradeoff between abstraction and performance, which is unfortunate but can be justified to some degree. However, there also seems to be a fine line between a compiler being powerful and helpful, for example by assisting the programmer in dealing with machine specific details, and a compiler trying to be too smart and getting in the way of the programmer. The virtual machine model used by CM Fortran [BHMS91] can be seen as a typical example of the latter [Chr91], as explained in more detail in Section 3.4. A programmer should not have to make this tradeoff when choosing a compiler, especially in a performance oriented field like scientific parallel computing. A compiler should try to assist the user in making some decisions, but it should also provide the user with convenient mechanisms to guide or override the compiler; such mechanisms are particularly important considering that parallelizing compiler technology is still in its infancy. Citing from a study about parallelizing different applications (including a molecular dynamics simulation) using the FX/Fortran parallelizing compiler [SH91]:

> It is worth noting that the available directives were sometimes found to be restrictive or incapable of expressing the exact information we wished to convey to the compiler. For example, the smallest entity for which we can tell the compiler not to check for dependences is an entire loop.

This observation has led to a trend away from completely parallelizing compilers, which try to extract parallelism from a sequential program without any user assistance, towards the development of more annotation oriented languages, which try to give the user a convenient interface for indicating parallelism. This approach is similar to the power steering paradigm [KMT91] used for loop transformations, where the compiler cannot always pick the best transformation, but it assists the user by (conservatively) testing correctness and performing the actual rewriting work.

### 2.4.1 Parallel Compilation Systems

There have been and still are numerous research projects in the area of compiling for parallel architectures. Early work in the field of compiling for distributed memory machines focussed on defining frameworks for nonlocal memory accesses [CK88] and data distributions [GB91, HA90, RS89]. For exploiting coarse-grained functional parallelism, high-level parallel languages such as Linda [CG89], Strand [FT90, FO90], and Delirium [LS91] have been defined.

Numerous compilation systems for exploiting fine-grained parallelism have been and are being built, which include Al [Tse90], Aspar [IFKF90], C*/Dataparallel C [HQL$^+$91, RS87], Crystal [LC91], Dino [RSW91], Id Nouveau [RP89], Mimdizer [SWW92], Oxygen [RA90], P$^3$C [GAY91], Pandore [APT90], Parafrase-2 [GB92], Paragon [CCRS91], Spot [SS90, Soc90], Superb [ZBG88], and Vienna Fortran [BCZ92]. While there is still much work to be done in this field in general, there has already been considerable success in the field of regular applications, and "second generation parallelizing compiler" has become a common term.

### 2.4.2 Fortran D

Fortran D is an SPMD (Single-Program Multiple-Data) style language developed by the distributed memory compiler group at Rice University and serves as a basis for this work. The following contains a very brief summary of its basic concepts, the complete language is described in detail elsewhere [FHK$^+$90]. Citing Hiranandani *et al.* [HKT92b]:

> FORTRAN D is the first language to provide users with explicit control over data partitioning with both data *alignment* and *distribution* specifications. The `DECOMPOSITION` statement specifies an abstract problem or index domain. The `ALIGN` statement specifies fine-grain parallelism, mapping each array element onto one or more elements of the decomposition. This provides the minimal requirement for reducing data movement for the program given an unlimited number of processors. The alignment of arrays to decompositions is determined by their subscript expressions in the statement; perfect alignment results if no subscripts are used.
>
> The `DISTRIBUTE` statement specifies coarse-grain parallelism, grouping decomposition elements and mapping them and aligned array elements to the finite resources of the physical machine. Each dimension of the decomposition is distributed in a block, cyclic, or block-cyclic manner or replicated.

### 2.4.3 Compilation Systems for Irregular Problems

Projects that have aimed at least to some degree towards compiler support for parallelizing irregular problems are the following.

**Kali** KALI [KMV90, MV90, KM91] is the first compiler system that supports both regular and irregular computations on MIMD distributed-memory machines. Programs written for KALI must specify a virtual processor array and assign distributed arrays to `BLOCK`, `CYCLIC`, or user-specified decompositions. Instead of deriving a computational decomposition from the data decomposition, KALI requires that the programmer annotates each parallel loop with an `ON` clause that maps loop iterations onto the processor array. Communication is then generated automatically based on the `ON` clause and data decompositions. An inspector/executor strategy as described in Section 2.3.2 is used for run-time preprocessing of communication for irregularly distributed arrays [KMSB90]. Major differences between KALI and the FORTRAN D compiler include KALI's mandatory `ON` clauses for parallel loops and FORTRAN D's support for alignment, collective communication, and dynamic decomposition.

**ARF** ARF is another compiler based on the inspector-executor paradigm. ARF is designed to interface Fortran application programs with the PARTI run-time routines described in Section 2.3.2 [WSHB91]. It supports `BLOCK`, `CYCLIC`, and user-defined irregular decompositions. The goal of ARF is to demonstrate that inspector/executors based on PARTI primitives can be automatically generated by the compiler.

### 2.4.4 Communication analysis

Determining communication requirements and satisfying them efficiently is critical for any parallel program. Eliminating redundant communication, message blocking and hoisting, and hiding communication delays are important optimizations, all of which are particularly difficult to perform for irregular problems. Our strategy for effective communication placement is based on an extensive dataflow framework, as outlined in Section 3.3 and described in more detail in Appendix B.

Dataflow Analysis is a common technique for reasoning at compile time about the run time behavior of the program concerning variable definitions and uses. The bulk of the work in this field has treated all variables as scalars, resulting in a very conservative analysis for array variables. More precise methods are based on representations of array subsets, like *data access descriptors* [Bal90] or *regular sections* [HK91].

The W2 compiler [GS90] for the Warp multiprocessor gathers information like the set of definitions reaching a basic block to exploit the fine-grain parallelism offered by the highly pipelined functional units. It is based on interval analysis [All70, Coc70] and computes information with array region granularity.

Granston and Veidenbaum combine flow and dependence analysis to detect redundant global memory accesses in parallelized and vectorized codes [GV91]. They assume that the program is already annotated with read/write operations. Their technique tries to eliminate these operations where possible, also across loop nests and in the presence of conditionals.

What appears to be lacking so far is a general approach towards analyzing the communication needs of a given program and determining when communication statements can be combined and hoisted. Developing a dataflow framework which provides this analysis and furthermore gives specific treatment for the access patterns induced by irregular applications is a substantial part of this proposal (Section 3.3).

### 2.4.5 Evaluation

As mentioned in Section 2.4.1, the area of compiling regular applications onto distributed memory machines has become very active, and much progress has been made. The formation of the High Performance Fortran Forum [Hig92], an ongoing standardization effort for commercial parallel Fortran compilers, is certainly an indication for this progress. High Performance Fortran (HPF) derives many of its underlying for FORTRAN D, which is the base language chosen for the extensions to be proposed here.

The body of work that focuses on irregular application is much smaller. In particular, there are no attempts to directly exploit the characteristics of underlying applications (like the positions of atoms in a protein); previous approaches are still based on access patterns (like a pairlist directly indicating the interaction partners for each atom) and try to determine data decompositions and communication optimizations *after* the access patterns of the program have been determined.

A compiler can not reasonably be expected to derive all locality aspects of the application underlying a given program. We do think, however, that there is a considerable potential for optimizations if the user has a convenient way to express locality information to the compiler. The main objectives of this dissertation are the design and evaluation of language extensions that provide such an interface and the development of the compiler analysis necessary to support these extensions.

## 2.5 The Operating System

Virtual or hardware supported single-address space systems can ease the task of parallel programming by eliminating separate address spaces and explicit communications. Examples of these systems are AMBER [CAL$^+$89], CLOUDS [RAK88], DASH [LLG$^+$90], IVY [LH89], MIDWAY [BZ91], MUNIN [CBZ91, KCZ92], ORCA [BT88], and PLATINUM [CF89]. They preserve sequential semantics by enforcing a consistency protocol, which can be lazy or eager, based on invalidations or updates. Munin supports several such protocols, the choice between them for each individual shared variable is guided by access pattern annotations provided by the user. These systems, however, are demand-driven and therefore limited in how much they can hide memory latency (by prefetching data before they are needed) or reduce data movement costs (by fetching entire blocks at once). They are limited in that they can only react to accesses to nonlocal memory; at best, they can maintain a history of past accesses and try to guess future patterns.

One problem where operating systems can assist in the implementation of irregular applications in particular is the task of load balancing, since some spatial and temporal locality is usually associated with the workload. Here information about the utilization of different processors can be helpful. However, the work done in this area has focussed on thread based parallelism [ELZ86, Luc88], typically even associated with distinct processes, instead of data parallelism.

## 2.6 The Hardware

Some hardware facilities that can be particularly useful for irregular applications are the following.

### 2.6.1 Low latency

Due to the typically very irregular access patterns, message blocking becomes more complicated than for regular applications [SHG92]. A low latency communication system makes this difficulty less critical.

### 2.6.2 General routing facilities

Again, due to indirect addressing and the associated irregular access patterns it is often difficult to constrain the communication to nearest neighbor communication channels, so a fast general router is advantageous.

### 2.6.3 Decoupling of control flow

A problem similar to the potential load imbalance across processors is an uneven workload *within* processors. This is also a common characteristic of irregular problems, where the fraction which a processor spends of its total computation time on a certain part of its assigned workload may vary. This may cause additional idling when running irregular problems on SIMD machines instead of MIMD machines.

The restricted control flow of pure SIMD programming has been addressed by several researchers. General simulators of MIMD semantics on SIMD machines have been implemented by Kuszmaul [Kus86] and Hudak *et al.* [HM88] on the Connection Machine and by Biagioni [Bia91] on the MasPar. These simulations are generally based on graph reduction interpreters for functional languages. Their performance tends to be scalable, but in absolute measures still below the speed of sequential workstations.

Philippsen *et al.* introduce two variants of a `FORALL` statement, a synchronous version and an asynchronous one [PT91a]. The *asynchronous* `FORALL` enables multiple threads of control to coexist. This can either be emulated using stacks of MASK bits, or it can be implemented directly in an MSIMD machine which contains multiple program counters. In either case, their proposal is mainly concerned with enabling the concurrent execution of both branches in IF-THEN-ELSE constructs.

*Loop flattening* [HK92] is one technique to overcome this limitation for loop nests with varying loop bounds, as proposed in Section 3.4. Loop flattening can also be used to process multiple array *segments* of different lengths per processor, as introduced in Blelloch's *V-RAM model* [Ble90]. Thus it can be viewed as a generalization of substituting direct addressing with indirect addressing as Tomboulian and Pappas did for computing the Mandelbrot set [TP90].

### 2.6.4 Fast scan operations

The inhomogeneous workload across processors generally associated with irregular problems calls for load balancing. Scan operations are one efficient way for determining the total workload and its distribution [Bia91, Ble90]. On architectures providing an embedded reduction tree, this operation can be done in $\mathcal{O}(\log P)$ cycles.

# 3 Research Plan

## 3.1 Overview

For establishing the thesis stated in Section 1, I am currently pursuing three closely related projects. These range from application oriented performance comparisons of different distribution strategies to developing a compiler strategy for analyzing communication characteristics of a given code to the development of architecture specific techniques for solving irregular problems. These undertakings are expected to lead to valuable insights towards

- Which requirements are given by irregular *applications*,
- How much support we can expect from the *compiler* for satisfying those, and
- Which difficulties we can expect from the *hardware* side.

These projects and their specific research contributions towards the thesis are summarized in Sections 3.2, 3.3, and 3.4, respectively; the Appendices A, B, and C contain more details for each of these.

## 3.2 Value-Based Decompositions

A standard problem in compiling for distributed memory multiprocessors is to determine which processors own which elements of a data array $X$ [CK88, Fox88, PRV87]. For simplicity of notation, we assume in the following that $X$ is three-dimensional; the extension to lower or higher dimensions is straightforward. Let

$$\begin{array}{rcll} \forall n \in \mathbb{N}, \mathbb{N}_n & = & \{i \mid i \in \mathbb{N}, 1 \leq i \leq n\}, & \\ I, J, K & \in & \mathbb{N} & \text{(the extent of } X \text{ in each dimension),} \\ \mathcal{S} & = & \mathbb{N}_I \times \mathbb{N}_J \times \mathbb{N}_K & \text{(the \textit{index set} of } X\text{), and} \\ P & \in & \mathbb{N} & \text{(the number of processors),} \\ \mathcal{P} & = & \{p_i \mid i \in \mathbb{N}_P\} & \text{(the set of \textit{processor labels}),} \\ \delta & : & \mathcal{S} \mapsto \mathcal{P} & \text{(the \textit{ownership mapping}).} \end{array}$$

The processor label of the owner of $X(i,j,k)$ is given by $\delta(i,j,k)$.

### 3.2.1 Data Distribution Mechanisms — Mapping Functions *vs.* Mapping Arrays

One important question is: how do we represent $\delta$ ? There are several tradeoffs to be made, regarding the locality properties of the data array and the complexity to represent and compute $\delta$.

- For static, regular mappings such as *BLOCK* (as shown in Figure 1) and *CYCLIC* , $\delta$ can be represented as a simple function involving some integer arithmetic based on just $I$, $J$, $K$, and $P$. Each processor can evaluate $\delta$ locally for all $(i,j,k) \in \mathcal{S}$. This function is typically hardwired into the compiler and the code it generates [FHK+90].
- For irregular mappings that can be represented in some compact format, for example because they map $\mathcal{S}$ into $P$ rectangular subdomains. In this case, $\delta$ can be represented as a function based on $I$, $J$, $K$, $P$, and some set $\mathcal{D}$ of dynamic parameters (such as the boundaries of the subdomains). $\mathcal{D}$ is typically small enough to be replicated across processors, so $\delta$ can still be evaluated locally. This kind of mapping is applicable if the data structure, here $X$, directly reflects spatial locality. Furthermore, a data structure with a fixed density, such as Variable Conductance Diffusion [Bia91], is preferable since in this case the memory requirements per spatial space unit are also fixed.

  These mapping functions can be further subdivided into several classes, ranging from restrictive but compact and fast to more general but larger and slower. Assuming that the processors are organized in a three-dimensional mesh, let $\mathcal{P} = \mathbb{N}_Q \times \mathbb{N}_R \times \mathbb{N}_S$, where $QRS = P$.

  - An *orthogonal* mapping function as seen in Figure 2 can be represented as a triplet of one-dimensional mapping functions $(\delta_x, \delta_y, \delta_z)$:

    $$\delta(i,j,k) = (\delta_x(i), \delta_y(j), \delta_z(k)).$$

    In this case, $\delta$ can be encoded by $Q + R + S - 3$ integers [Bia91], which is typically $\mathcal{O}(\sqrt[3]{P})$ for a 3-D index space.
  - A *hierarchical* mapping function, as illustrated in Figure 3, can also be represented as a triplet of mapping functions $(\delta_x, \delta_y, \delta_z)$. These, however, are not all one-dimensional any more; instead, it is

    $$\delta(i,j,k) = \delta_x(i, \delta_y(j, \delta_z(k))).$$

    Here $\delta$ can be represented with $S - 1 + (R-1)S + (Q-1)RS = P - 1$ integers [CHMS92], see also Section A.4.1.
  - A *general* mapping function cannot be represented as a simple composition of subfunctions; *i.e.*, we cannot decouple any of the dimensions from each other. A special case here are *recursive* mapping functions like the orthogonal recursive bisection shown in Figure 4, which can still be encoded in $P - 1$ integers [Han89].

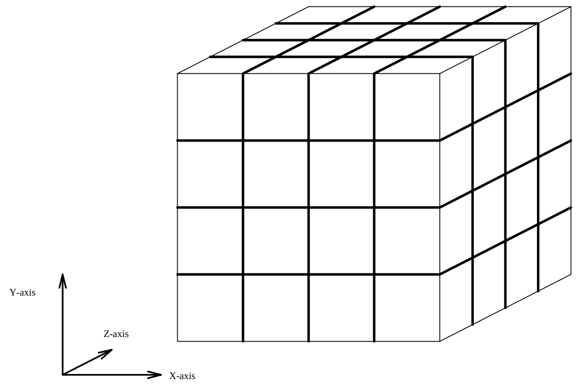

**Figure 1** Example of a *BLOCK* decomposition for $P = 64$ processors.

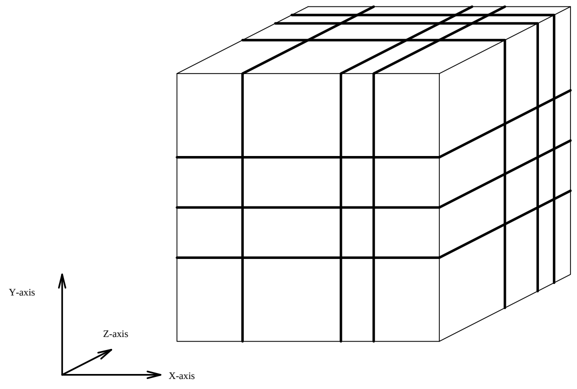

**Figure 2** Example of an orthogonal mapping for $P = 64$ processors.

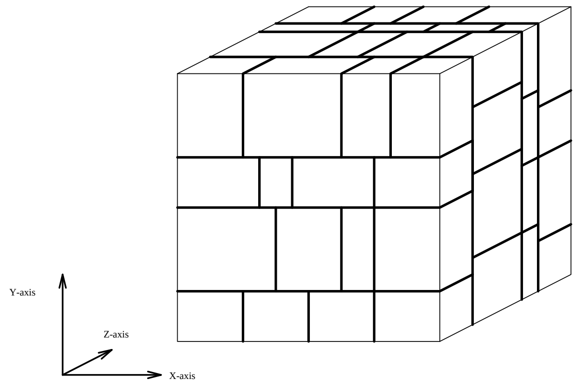

**Figure 3** Example of a hierarchical mapping for $P = 64$ processors.

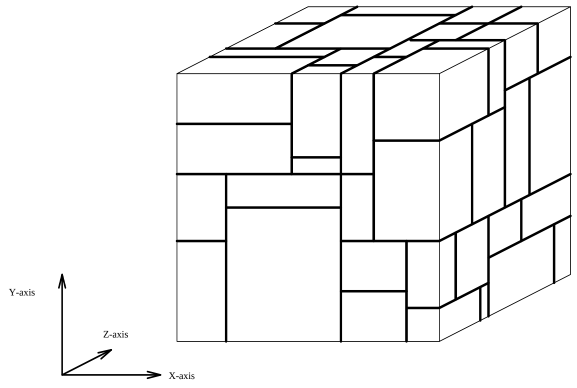

**Figure 4** Example of recursive bisective mapping for $P = 64$ processors.

The encoding sizes given here for the mapping functions are worst case lower bounds; we can typically trade representation storage for evaluation speed and precompute the functions to some degree, resulting in larger representations.

- If the way data are stored in $X$ does not reflect their spatial locality, then $\delta$ can probably not be represented in some compact form. In this case, a *mapping array* that contains the precomputed values of $\delta(i,j,k)$ for all $(i,j,k) \in \mathcal{S}$ is appropriate. Since now $\delta$ is encoded in an $\mathcal{O}(N)$ data structure (where $N = IJK$ is the problem size), this typically implies that $\delta$ itself has to be (regularly) distributed. This may result in an extra communication step to first determine the ownership and then fetch the data from the owner as usual.

### 3.2.2 Index *vs.* Value-Based Decompositions

Section 3.2.1 gave an overview about different representations of the ownership mapping $\delta$. The choice of the appropriate representation for $\delta$, however, depends on how the data themselves are stored, *i.e.*, which data structure we use for $X$. In PIC (Particle-In-Cell) codes, for example, we can store the data linked to a particle in a data structure whose elements are associated with spatial locations directly. This enables a fast neighbor-lookup, but it forces us to provide sufficient space for the maximum particle density; *i.e.*, if we quantize spatial space into $l$ locations and can bound the number of particles per location by $\rho$, then we have to provide $l\rho d$ bytes storage, where $d$ is the number of bytes needed to describe a particle. However, it is sufficient to just store for each spatial location a pointer into another data structure where one linked list of actual data is kept for each spatial location. If $n$ is an upper bound on the total number of particles, then $lp + n(d+p)$ bytes are sufficient, where $p$ is the size of a pointer in bytes. For $n \ll l\rho$ and $1 \ll \rho$, which is typically the case, this can be a substantial savings in storage, at the expense of using linked lists.

Revisiting the definition of $\delta$, we notice that it maps array element *indices* to processors. On the one hand, this is certainly appropriate, since a program accesses array data by their indices. On the other hand, an important objective in mapping data to processors is to achieve high locality, not only in terms of indices, but also regarding the underlying physical problem. This typically implies that there has to be a correspondence between array indices (*index locality*) and spatial coordinates (*spatial locality*), with the disadvantage of increased memory requirements as mentioned above for PIC codes.

The approach developed in this thesis tries to resolve this conflict by relaxing this correspondence requirement and having the compiler keep track of spatial locality independent of index locality. A *value-based decomposition*, *VDECOMP*, is based on the values of an array, $X$, at a certain point. Internally, *VDECOMP* may be represented as a mapping array, but this should be opaque to the user (see Appendix A.6 for details). In other words, the mapping function $\delta$ still effectively maps array indices to processors, but the *definition* of $\delta$ is based on a snapshot of values. As usual, other arrays may be aligned with *VDECOMP*; *e.g.*, after deriving a decomposition *VDECOMP* from a coordinate array $X$, we may align not only $X$ but also velocities $V$ and forces $F$ to *VDECOMP*.

### 3.2.3 Contributions

The development and benchmarking of different value-based decomposition schemes and their comparison with non-value-based approaches from the performance point of view is an important validation for determining how far these schemes can and should be implemented in a compiler.

Although the concept of value-based decompositions was developed with simple arrays as underlying data structures, the thesis will also contain an analysis of how well this concept applies to the more advanced data structures used in hierarchical solvers [Ben75, FB74, LW82]. On the practical side, this involves the parallelization of a Molecular Dynamics code (GROMOS). This is accompanied with a comparison between FORTRAN D and low level message passing code, IPFORTRAN, which still operates in the local name space of message passing code, but provides language constructs for easy access of non-local values [BCS91]. The dominating difficulties addressed here are of type 1 and 3 (see Section 1.1). As described in Appendix A, this project already has led to valuable insights into which of the language concepts well proven for regular problems carry over easily into the irregular world and which concepts have to be modified or extended [CHK+92, CHMS92].

## 3.3 Communication Analysis for Irregular Problems

One issue related to value-based decompositions, and to irregular decompositions in general, is how to generate the necessary communication for accessing the data distributed this way. We decided to use the Parti communication routines which are described in Section 2.3.2. The remaining question is how to generate calls to Parti routines and where to place these calls. For example the unstructured mesh solver of which Figure 5 shows a simplified and abstracted version has several opportunities for combining communication and hoisting communications out of loops (see Section B.7). However, to derive a good communication placement as the one shown in Figure 6 is nontrivial.

The approach taken in this thesis is based on a dataflow framework [HKK+92] which bears similarities to classical techniques such as common subexpression elimination, loop invariant code motion, and dead code elimination. The framework provides a basis for determining at compile time

- Where communication schedules are to be generated,
- Where gather, scatter, and accumulate operations are to be placed, and
- When combined or incremental schedules may be employed.

In our data flow analysis, some of the variables reflect inherent properties of the analyzed program, while others calculate the results of heuristics we employ in order to produce the gather and scatter operations. Our heuristics aim to

- Exploit situations where we can reuse communication schedules, and to
- Remove extraneous communication by combining and hoisting gather, scatter and accumulate procedures.

The domain of this framework is the *program flow graph* (see Appendix B.1.1), as shown in Figure 7 for the example from Figure 6. For the example code shown in Figure 5, the framework would derive the result shown in Figure 8 (see also Section B.7), which corresponds to the code shown in Figure 6. The dataflow framework aims at providing good information about which data are needed at which points in the code, along with information about which live off-processor data are available. We generate *combined* communication schedules (Section 2.3.2) that combine off-processor data for several indirect references, possibly contained in different loops, and we generate *incremental* schedules to obtain only those off-processor data that are not already requested by a given set of pre-existing schedules. This gives the runtime support needed to combine and hoist gather, scatter, and accumulation operations. Eliminating redundant data movements in the communication schedules is achieved by using a hash table.

### 3.3.1 Contributions

A major part of this thesis is the design and implementation of a data flow framework for compile time reasoning about irregular array access patterns that occur when using value-based decompositions [HKK+92]. The generated code should make effective use of the scatter/gather operations provided by the Parti system (see Section 2.3.2) and maximize message blocking and data reuse, as described in Section B. This mainly addresses Difficulties 2 and 4 in Section 1.1.

The implementation work will be done within the FORTRAN D MIMD distributed memory compiler prototype developed at Rice. The validation is based on comparing, for a test suite of programs, how well the data flow framework does in communication placement compared to hand crafted code, and how much of its power is actually needed in these programs. So far, the prospective test suite includes programs covering

- molecular dynamics (GROMOS, see Section A.1),
- unstructured meshes [Mav91], and
- sparse matrices.

Beyond working with these source codes, other applications (whose source codes are not available or not written in Fortran) will be evaluated at least theoretically [HKK+91].

## 3.4 MIMD *vs.* SIMD for Irregular Problems

In the process of parallelizing scientific programs, it is common to find loop nests in which the outer loop can run in parallel but the amount of computation in the inner loop varies for different iterations of the outer loop. A typical case is the calculation of nonbonded forces as described in Section A.2: an outer loop steps through atoms, each of which is associated with a varying number of partners enumerated in a pairlist through which we step in an inner loop. This causes a load balancing problem because the outer loop iterations have to be partitioned among the processors in such a manner that each processor has a roughly equal amount of work to do (Difficulty 1 in Section 1.1). Load balancing is a difficult problem in itself which has been frequently addressed in the literature [BBHL90, BB87, Bok81, DG90, FKW86, FJL+88, HS91a, SHT+92]. Once this problem (including related issues like data locality) is solved, we can usually expect good performance when running such a loop nest on a shared-memory or distributed-memory MIMD (Multiple Instruction, Multiple Data) machine.

However, this kind of loop nest causes special problems for SIMD (Single Instruction, Multiple Data) architectures because of the restricted control flow on these machines [Brä89]. If the number of iterations of the inner loops varies from one outer loop iteration to the next, as for example in the code shown in Figure 9, then the restriction to a common program counter makes a naïve SIMD implementation inefficient. As observed in a case study implementing an image processing algorithm on the Massively Parallel Processor [WLR90, page 143]: "... the complexity of each iteration in the SIMD environment is dominated by the largest region in the image. This is due to the fact that the synchronous execution of instructions forces each processor to either perform the operation or wait in an idle state until all processors have completed the operation. " To overcome this limitation, we propose a transformation which we call *loop flattening* [HK92]. Roughly speaking, loop flattening amounts to lifting the innermost loop body up into the outer, parallel loop by merging the control of the inner loops with the control of the outer loop, as shown in Figure 10. Further details on this can be found in Appendix C.

### 3.4.1 Contributions

The development of the loop flattening concept, which is a loop transformation strategy to overcome certain control flow limitations on SIMD architectures [HK92], addresses the problem of providing architecture independent compiler support for parallelizing irregular problems. These limitations emerge within nested loops if the number of iterations of the inner loop varies between different iterations of the outer loop. This situation occurs frequently in irregular problems like the calculation of non-bonded forces between atoms, as described in Section A.2.

After load balancing, this kind of nested loop with varying inner loop bounds usually does not pose any problem for sequential or MIMD machines; for SIMD machines, it can be problematic, since the restriction to a common program counter means for a direct implementation of such a nested loop that for *each* parallel outer loop execution, all processors have to wait for the most time consuming inner loop before advancing to the next outer loop iteration. This bottleneck, caused by Difficulty 1, can be avoided by flattening loop nests, which is described in more detail in Section 3.4.

A high level outline of how to automate this transformation will be part of this thesis, as well as a practical case study on how much loop flattening can improve the performance of SIMD machines for certain kinds of irregular problems. However, since the implementation part of this thesis focuses on the FORTRAN D MIMD distributed memory compiler prototype, a general implementation of automated loop flattening should not be part of this work.

## 3.5 Summary

To summarize, the research to be covered in the thesis includes the following.

- The extension and development of machine independent parallel language constructs supporting irregular problems, in particular for providing value-based decompositions.
- The evaluation of their usability with respect to different irregular applications.
- The implementation of a subset of these language extensions into the existing FORTRAN D prototype compiler for *MIMD* distributed memory machines, potentially covering interprocedural analysis as well. This includes the implementation of support for mapping arrays, different mappers to automatically determine the values of the mapping arrays (see Section 3.2.1), and a dataflow framework for placing calls to PARTI communication routines (as described in Section 3.3).
- Experimenting with these language extensions and comparing the compiler generated code and its performance with handcrafted parallel code.
- The development and evaluation of the loop flattening transformation, which on SIMD machines improves the performance of loops with varying inner bounds, which are typical for irregular problems.

# 4 Conclusions

The projects described in Sections 3.2, 3.3, and 3.4 focus on different aspects of the same problem, namely how to solve irregular applications efficiently with parallel machines.

Furthermore, they all focus on the issue of how much support a compiler can give for these applications. The results so far seem to indicate that it is feasible to design high level language support similar to the support existing for regular problems and to implement it at reasonable costs. An important example of this is the concept of the value-based decomposition based on the exploitation of spatial locality of the underlying physical problem, as introduced in Section 3.3 and in Appendix A.

After having decided on a certain decomposition, communicating the right data at the right time and place is still a difficult, yet crucial task for parallelizing irregular problems. The PARTI primitives are valuable tools for the first part of the problem, namely for determining where to find which data and for efficient data exchange. The dataflow framework presented in Section 3.3 and Appendix B is designed to attack the second part of the problem, namely enabling the compiler to make good use of these primitives without further advice by the user. We believe our approach to be effective for a wide range of interesting problems, as illustrated with an explicit unstructured mesh solver.

However, there is obviously still room for further improvement and generalization of the underlying theory. For example, the loop flow graph representation as described in Subsection B.1.1 seems to be convenient and well suited for a first implementation, but has some unsatisfactory aspects from the computer science point of view, such as the ad hoc distinction between loops that directly enclose indirect references and other loops. This distinction simplifies the use of portions (see Section B.1.2), which contain references to iteration ranges, but it constrains the generality of the framework. Depending upon which heuristic we use, this distinction may, for example, limit the loop nesting depth out of which we hoist communication statements. A more general representation of the program could be based on basic blocks instead of inner loops. A complication then arising would be the loss of an implicit iteration space for each irregular reference. To determine which references actually constitute a particular portion, one can use *slicing* [HRB90]. (This is in fact already part of our current implementation design, the representation of portions in the fixed format introduced in Section B.1.2 was mainly used for illustration purposes).

Another relatively straightforward extension is to break each communication call up into their matching send/receive pairs and then place these components such as to overlap communication and computation as much as possible. For example, when we break each GATHER into a $\text{GATHER}_{send}$ and a $\text{GATHER}_{recv}$ (Section B.4), we can place the $\text{GATHER}_{send}$ as currently determined by *GATH*, and place the matching $\text{GATHER}_{recv}$ statement by delaying it along each path starting at the $\text{GATHER}_{send}$ statement until we reach the first use of the communicated data. (This assumes blocking receives; with non blocking receives, we would place the $\text{GATHER}_{recv}$ together with the $\text{GATHER}_{send}$, and delay a $\text{GATHER}_{wait}$ statement instead to make sure the data are available before we try to use them). A further refinement that would be useful for block structured irregular problems, for example, is to allow data exchanges between just *subsets* of processors, instead of requiring a global coordination whenever the primitives are called as mentioned in the introduction.

One might also consider relaxing the static ownership concept for data accessed via indirection arrays. For example, it might occur that a processor $p$ computes some data that are owned by processor $q$, but the next use of the data is on processor $r$. In the current owner computes framework, we would first scatter the data from $p$ to $q$ and then gather them from $q$ to $r$, which could be replaced by a single "scatter-gather" from $p$ to $r$. This, however, would add an additional degree of complexity to the theoretical framework and the underlying communication primitives.

Finally, one might consider pruning the framework down towards regular applications, where the same need for blocking and combining communication arises. So far, this is typically handled with local code transformations based on dependence analysis, but there is no inherent reason for not applying data flow analysis here as well. This could be done by basically using the framework proposed here and replacing portions with classical array regions.

The loop flattening transformation described in Section 3.4 is an attempt to overcome certain limitations when using a SIMD computer for solving irregular problems without going as far as trying to achieve general MIMD semantics on SIMD machines. Loop flattening was designed to ease some particular SIMD restrictions without introducing any overheads; however, it supports a programming style that seems to be preferable on current SIMD machines even when running regular applications [HK92]. This rather surprising result suggests that flattened loops make it easier for compilers to derive the information they need for performing certain optimizations, such as pruning out virtual processor layers for individual statements whenever possible. This transformational approach might possibly be carried over into other situations where the restrictiveness of the SIMD model degrades overall performance.

So far, there does not seem to be a clear limit to what support a compiler, when given the proper analysis, can give the scientist programming an irregular problem. It is my hope that this dissertation will advance the field in this particular area by some steps within the FORTRAN D framework and will also indicate possible future directions for increasing the compiler support for parallelizing this important class of applications even further, for example by providing some support for the conversion of data structures as outlined in Section A.4.

## References

[All70] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 1970.

[APT90] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Bad87a] S. B. Baden. *Run-Time Partitioning of Scientific Continuum Calculations Running on Multiprocessors.* PhD thesis, Lawrence Berkeley Laboratory, University of California, 1987.

[Bad87b] S. B. Baden. Very large vortex calculations in two dimensions. In *Vortex Methods*, volume 1360 of *Lecture Notes in Mathematics*, Los Angeles, CA, May 1987. Springer-Verlag.

[Bad91] S. B. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 12(1):145–157, 1991.

[Bad92] S. B. Baden. Lattice parallelism: A parallel programming model for manipulating localized non-uniform scientific data structures. In *Intel Super-*

*computer University Partners Conference*, Timberline Lodge, Mt. Hood, OR, April 1992.

[Bal90] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[BB87] M. J. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[BBHL90] T. Bemmerl, A. Bode, O. Hansen, and T. Ludwig. A testbed for dynamic loadbalancing on distributed memory multiprocessors. PUMA Working Paper 14, Technical University Munich, München, Germany, August 1990.

[BBO$^+$83] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[BCS91] B. Bagheri, T. W. Clark, and L. R. Scott. IPfortran (a parallel extension of Fortran) reference manual. Research Report UH/MD–119, Dept. of Mathematics, University of Houston, 1991.

[BCZ92] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–516, 1975.

[BH86] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 1986.

[BHMS91] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[Bia91] E. S. Biagioni. *Scan Directed Load Balancing.* PhD thesis, University of North Carolina at Chapel Hill, 1991.

[Ble90] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[Bok81] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[Brä89] T. Bräunl. Structured SIMD programing in Parallaxis. *Structured Programming*, 10(3):121–132, 1989.

[BS90] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. ICASE Interim Report 13, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[BSGM90] H. Berryman, J. Saltz, W. Gropp, and R. Mirchandaney. Krylov methods preconditioned with incompletely factored matrices on the CM-2. *Journal of Parallel and Distributed Computing*, 8:186–190, 1990.

[BT88] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[CAL$^+$89] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CCRS91] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[CF89] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with Platinum. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CHK$^+$92] T. W. Clark, R. v. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott. Evaluating parallel languages for molecular dynamics computations. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[CHMS92] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization strategies for a molecular dynamics program. In *Intel Supercomputer University Partners Conference*, Timberline Lodge, Mt. Hood, OR, April 1992.

[Chr91] P. Christy. Virtual processors considered harmful. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CM90] T. W. Clark and J. A. McCammon. Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architectures. *Computers & Chemistry*, 14(3):219–224, 1990.

[CMS91] T. W. Clark, J. A. McCammon, and L. R. Scott. Parallel molecular dynamics. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[Coc70] J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 1970.

[Dah90] D. Dahl. Mapping and compiled communication on the connection machine system. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[DG90] F. Dehne and M. Gastaldo. A note on the load balancing problem for coarse grained hypercube dictionary machines. *Parallel Computing*, 1990.

[DMS$^+$92] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*. AIAA, January 1992.

[EHL77] J. Eastwood, R. Hockney, and D. Lawrence. PM3DP – the three dimensional periodic particle-particle/ particle-mesh program. *Computer Physics Communications*, 19:215–261, 1977.

[ELZ86] D. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[FB74] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[FHK$^+$90] G. C. 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. Revised April, 1991.

[FJL$^+$88] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Multiprocessors.* Prentice-Hall, 1988.

[FKW86] G. C. Fox, A. Kolowa, and R. Williams. The implementation of a dynamic load balancer. In *Proceedings of the Second Hypercube Microprocessors Conference*, pages 114–121, Knoxville, TN, September 1986.

[FO90] I. Foster and R. Overbeek. Bilingual parallel programming. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[Fox88] G. C. Fox. Domain decomposition in distributed and shared memory environments. In *Proceedings of Supercomputing '88*, pages 1042–1073, Orlando, FL, November 1988.

[Fox91] G. Fox. Parallel problem architectures and their implications for portable parallel software systems. CRPC Report CRPC-TR91120, Center for Research on Parallel Computation, Syracuse University, February 1991.

[FT90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[GAY91] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[GB88] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[GB91] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[GB92] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulation. *Journal of Computational Physics*, 73(325), 1987.

[GS90] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[GV91] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[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.

[Han89] R. v. Hanxleden. Parallelizing dynamic processes. Master's thesis, Dept. of Computer Science, The Pennsylvania State University, August 1989.

[HC88] P. N. Hilfinger and P. Colella. FIDIL: A language for scientific programming. Technical report, Lawrence Livermore National Laboratory, 1988.

[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.

[Hig92] *Proceedings of the High Performance Fortran Forum*, Houston, TX, January 1992.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HK92] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[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.

[HKK+92] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the*

*Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[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.

[HKT92a] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[HKT92b] 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.

[HM88] P. Hudak and E. Mohr. Graphinators and the duality of SIMD and MIMD. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HQL$^+$91] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HS91a] R. v. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[HS91b] R. v. Hanxleden and L. R. Scott. Parallelizing dynamic processes on message passing architectures. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, March 1991.

[HS92] R. v. Hanxleden and L. R. Scott. Correctness and determinism of parallel Monte Carlo processes. *Parallel Computing*, 18:121–132, 1992.

[HT84] T. Hoshino and K. Takenouchi. Processing of the molecular dynamics model by the parallel computer PAX. *Computer Physics Communications*, 31(4), 1984.

[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.

[KCZ92] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KM91] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMSB90] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KMT91] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[KU76] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):159–171, January 1976.

[Kus86] B. C. Kuszmaul. Simulating applicative architectures on the Connection Machine. Master's thesis, Massachusetts Institute of Technology, 1986.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LLG+90] D. Lenoski, J. Laudon, K Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[LS91] S. Lucco and O. Sharp. Parallel programming with coordination structures. In *Conference Record of the Eighteenth ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[Luc88] B. J. Lucier. Performance evaluation for multiprocessors programmed using monitors. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 16 of *SIGMETRICS Performance Evaluation Review*, 1988.

[LW82] G. S. Lueker and D. E. Willard. A data structure for dynamic range queries. *Inf. Proc. Lett.*, 15:209–213, 1982.

[Mas91] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran Reference Manual*, 1991.

[Mav91] D. Mavriplis. Three dimensional unstructured multigrid for the euler equations. Technical report, Institute for Computer Application in Science and Engineering, 1991.

[McC87] J. A. McCammon. Computer-aided molecular design. *Science*, 238:486–491, October 1987.

[MH87] J. A. McCammon and Stephen C. Harvey. *Dynamics of proteins and nucleic acids.* Cambridge University Press, Cambridge, MA, 1987.

[MPB91] F. Müller-Plathe and D. Brown. Multi-colour algorithms in molecular simulation: Vectorisation and parallelisation of internal forces and constraints. *Computer Physics Communications*, 64:7–14, 1991.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[MSS+88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[MV90] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[PRV87] M. L. Patrick, D. A. Reed, and R. G. Voigt. The impact of domain partitioning on the performance of a shared multiprocessor. *Parallel Computing*, 5:211–217, 1987.

[PT91a] M. Philippsen and W. F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[PT91b] M. R. S. Pinches and D. J. Tildesley. Large scale molecular dynamics on parallel computers using the link-cell algorithm. *Molecular Simulation*, 6:51–87, 1991.

[RA90] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[RAK88] U. Ramachandran, M. Ahamad, and Y. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[RCB77] J. Rycaert, G. Ciccotti, and H. J. C. Berendsen. Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-Alkanes. *Journal of Computational Physics*, 23:327–341, 1977.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the ACM SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[RS87] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of* *the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[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.

[RSW91] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[SBW90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. ICASE Report 90-59, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[SCMB90] 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(2):303–312, 1990.

[SH91] J. P. Singh and J. L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991.

[SHG92] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical N-body methods for multiprocessor architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[SHT$^+$92] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. Load balancing and data locality in hierarchical N-body methods. Technical Report CSL-TR-92-505, Stanford University, January 1992.

[SM90] T. P. Straatsma and J. Andrew McCammon. ARGOS, a vectorized general molecular dynamics program. *Journal of Computational Chemistry*, II(8):943–951, 1990.

[SM91] J. Shen and J. A. McCammon. Molecular dynamics simulation of Superoxide interacting with Superoxide Dismutase. *Chemical Physics*, 158:191–198, 1991.

[Soc90] D. Socha. Compiling single-point iterative programs for distributed memory computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SPBR91] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multicomputers. ICASE Report 91-12, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[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.

[SWW92] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and data decomposition. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. Elsevier, Amsterdam, The Netherlands, 1992.

[Thi91] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, 1991.

[TP90] S. Tomboulian and M. Pappas. Indirect addressing and load balancing for faster solutions to the Mandelbrot set on SIMD architectures. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 443–450, College Park, MD, October 1990.

[Tse90] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the ACM SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[WLR90] M. Willebeek-LeMair and A. P. Reeves. Solving nonuniform problems on SIMD computers: Case study on region growing. *Journal of Parallel and Distributed Computing*, 8:135–149, 1990.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[WSHB91] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

# A Value-Based Decompositions in Molecular Dynamics

## A.1 Molecular dynamics

First developed for simulating atomic motion in simple liquids, molecular dynamics is used routinely to simulate biomolecular systems and to obtain various kinetic, thermodynamic, mechanistic, and structural properties [McC87, MH87]. In molecular dynamics, the motion of each atom, represented as a point mass, is determined by the forces exerted on it by other atoms.

Molecular dynamics algorithms commonly iterate over the sequence:

1. *Calculate bonded and nonbonded forces* on each atom as the analytical gradient of a potential-energy function of the atom positions.

   The calculation of the nonbonded forces between atoms has $\mathcal{O}(N^2)$ time complexity and typically constitutes over 90% of the overall computational effort [CMS91]. This calculation is also particularly difficult to parallelize, since the data access patterns are highly irregular, as illustrated in Section A.2. However, it has a high potential for reducing non-local data accesses by taking advantage of the spatial locality of the computation.

2. *Integrate Newton's equations of motion* to determine the new atomic momenta and positions. By removing uninteresting, high-frequency motions, larger timesteps

can be taken to improve overall efficiency [MPB91, RCB77]. This usually involves the constraining of some molecular motions.

3. *Save data* as appropriate for post analysis.

Several program suites have been developed for the dynamic modeling of biomolecules, CHARMM [BBO+83] and GROMOS [GB88] being two commonly used packages. GROMOS (GROningen MOlecular Simulation) serves as a basis for a long-term parallelization project that originated at the University of Houston and is now carried out jointly by the University of Houston and Rice University. GROMOS provides programs for the simulation of biological molecules (and arbitrary molecules) using molecular dynamics or stochastic dynamics. In addition, energy minimization and analysis programs are provided. The approximately 127 files comprising GROMOS consist of about 74,000 lines of FORTRAN77, comments included.

## A.2 The Nonbonded Force Calculation

If the number of recent publications in this area is any indication, then the parallelization of the nonbonded force computations is challenging even from the human programmer's point of view. The following contains an evaluation of this problem from the compiler's point of view, starting with some background and brief descriptions of the experiences gained when parallelizing the nonbonded force kernel manually. More detailed accounts on other aspects of this project, like the parallelization of the pairlist generation or the calculation of bonded forces, can be found elsewhere [CM90, CMS91, CHK+92, CHMS92].

The GROMOS code approximates nonbonded forces by calculating them only for atom pairs that are within a certain *cutoff* radius of each other. These pairs are stored in a pair list that is updated in regular intervals, where each atom is identified by a *gan* (*global atom number*). In the original code, this pair list is represented by two arrays, *INB* and *JNB*. *INB*$(I)$ gives the number of partners of atom $I$, and *JNB* can be thought of as a concatenation of lists of partners, one list for each atom. We also introduce the arrays *firstJ* and *lastJ*, so that the array section *JNB*(*firstJ*$(I)$ : *lastJ*$(I)$) gives the list of partners of atom $I$. Obviously, *INB*$(I)$ = *lastJ*$(I)$ − *firstJ*$(I) + 1$.

According to Newton's Third Law, for each force exerted by an atom $A$ on an atom $B$, atom $B$ exerts an equal but opposite force on atom $A$. We therefore can cut the number of force calculations in half by storing each atom pair only once, for example by storing only partners with a higher atom index. The resulting sequential version for $N$ atoms is shown in Figure 11, where we assume that the force array $F$ is initialized to 0 (similarly in the following code samples). Note also that in practice the forces $F$ and the positions $X$ are vectors in $\mathbb{R}^3$.

A general strategy towards parallelizing this kernel is to make each processor responsible for computing the nonbonded forces for a certain set of atoms. For processor $p$, let *Owned*$(p)$ be the set of atoms for which $p$ is responsible, and let *Stored*$(p)$ be the set of atoms whose data are stored on $p$. Note that *Owned*$(p)$ and *Stored*$(p)$ are not necessarily the same (see Section A.3). Before describing some different approaches in more detail, however, we will introduce some formal tools for analyzing their locality characteristics.

### A.2.1 The computational cost

Let $\mathit{JNBL}(I,:)$ represent all of the partners of atom $I$ whose atom number is *greater* than $I$:

$$\mathit{JNBL}(I, 1 : \mathit{INB}(I)) = \mathit{JNB}(\mathit{firstJ}(I) : \mathit{lastJ}(I)).$$

This leads to a predicate that indicates whether an atom pair $(I, K)$ is stored in the pair list or not:

$$\mathit{isPair}(I,J) = \begin{cases} 1 & \text{if } \exists M,\ \mathit{JNBL}(I,M) = J, \\ 0 & \text{otherwise;} \end{cases}$$

$$I > J \Longrightarrow \mathit{isPair}(I,J) = 0 \qquad \textit{(Newton's Third Law)} \qquad (1)$$

Based on *isPair*, *INB* can be expressed as well as its dual, *Partners*:

$$\mathit{INB}(I) = \sum_{J=1}^{N} \mathit{isPair}(I,J), \qquad \text{and} \qquad \mathit{Partners}(J) = \sum_{I=1}^{N} \mathit{isPair}(I,J).$$

Since each processor computes the forces for the atoms it owns, we can approximate the workload of processor $p$ with

$$\mathit{Pairs}(p) = \sum_{I \in \mathit{Owned}(p)} \sum_{J=1}^{N} \mathit{isPair}(I,J) = \sum_{I \in \mathit{Owned}(p)} \mathit{INB}(I).$$

In terms of averages, we have

$$\mathit{INB}_{ave} = \frac{1}{N} \sum_{I=1}^{N} \mathit{INB}(I),$$

$$\mathit{Pairs}_{ave} = \frac{1}{P} \sum_{p=1}^{P} \mathit{Pairs}(p) = \frac{1}{P} \sum_{p=1}^{P} \sum_{I \in \mathit{Owned}(p)} \mathit{INB}(I) = \frac{1}{P} \sum_{I=1}^{N} \mathit{INB}(I) = \frac{N \times \mathit{INB}_{ave}}{P}.$$

Therefore, the overall *computational cost* is given by

$$T_{comp} \propto \max_{p=1}^{P} \mathit{Pairs}(p) = \max_{p=1}^{P} \sum_{I \in \mathit{Owned}(p)} \mathit{INB}(I).$$

This results in a typical load balancing problem, where the goal is to lower $T_{comp}$ down to $T_{ideal} \propto \mathit{Pairs}_{ave}$; *i.e.*, we want to achieve for all $p$:

$$\sum_{I \in \mathit{Owned}(p)} \mathit{INB}(I) \approx \frac{N \times \mathit{INB}_{ave}}{P}.$$

### A.2.2 The data locality

For deciding if and how data should be distributed across processors, an important measure is to what degree a processor $p$ accesses data (as reads or writes) that are not in $\mathit{Owned}(p)$. For that purpose, we determine the number of writes from $p$ to the force $F(K)$:

$$\mathit{Writes}(p,K) \;=\; \sum_{I \in \mathit{Owned}(p)} \mathit{isPair}(I,K) + \begin{cases} \mathit{INB}(K) & \text{if } K \in \mathit{Owned}(p), \\ 0 & \text{otherwise.} \end{cases}$$

The sum of these across all processors, broken up into writes by owners and other writes:

$$\begin{aligned} \mathit{OwnedWrites}(K) &= \sum_{p=1}^{P} \sum_{K \in \mathit{Owned}(p)} \mathit{Writes}(p, K), \\ \mathit{NotOwnedWrites}(K) &= \sum_{p=1}^{P} \sum_{K \notin \mathit{Owned}(p)} \mathit{Writes}(p, K), \\ \mathit{AllWrites}(K) &= \mathit{OwnedWrites}(K) + \mathit{NotOwnedWrites}(K). \end{aligned}$$

It turns out that *AllWrites* is independent of how we distribute ownership across processors:

$$\mathit{AllWrites}(K) = \mathit{Partners}(K) + \mathit{INB}(K).$$

To illustrate the irregularity of a typical molecular dynamics problem, Figure 12 contains a plot of *AllWrites* and *INB* for the bovine superoxide dismutase molecule ($SOD$) and $O_2^-$ in water, a total of 6968 atoms. SOD is a catalytic enzyme composed of two identical subunits, each with 151 amino-acid residues and two metal atoms [SM91]. Note that the data show a very high irregularity even though a *smoothing factor* of 20 has been applied, *i.e.*, each data point on the plot represents the average of 20 actual data points. However, we can still see the (strongly non-monotonic) decrease of $\mathit{INB}(K)$ as $K$ increases, which is due to Equation 1.

## A.3 Data Replication — The UHGromos Approach

The first parallelization of Gromos performed at the University of Houston was based on IPFortran [BCS91], which is an SPMD style language like Fortran D. The key concept of IPFortran is to provide a better abstraction for interprocessor communication than simple message-passing [Hoa85]. Unlike for the *global* memory model used by Fortran D, here the variables are implicitly *local* to each processor; thus, $X$ on processor 1 may have a different value from $X$ on processor 2. Nonlocal accesses are denoted by the @ operator, so $A(i)@j$ means the $i$-th element of array $A$ on processor $j$. Note that only the processor *using* a nonlocal value must reference it, and that the reference may be made within a larger expression. This is in contrast to message-passing languages, which require matching but separate "send" and "receive" operations. Global reductions are also supported; for example, $+\{X\}$ denotes the sum of all values of $X$ on all processors.

The objective for this first parallel version of Gromos, called UHGromos, was to achieve good performance for medium numbers of Atoms ($N \leq 10000$) and medium numbers of processors ($P \leq 128$), with a *minimal impact* on the original, sequential code. This goal was certainly achieved; for example, for $P \geq 16$ the performance on an iPSC 860 was better than on a single processor CRAY 2 [CMS91].

### A.3.1 The nonbonded force calculation in UHGromos

In UHGromos, the nonbonded force calculation is parallelized by having each processor $p$ compute the forces for a *continuous range of atoms indices*:

$$\mathit{Owned}(p) = \{I \mid \mathit{firstI}(p) \leq I \leq \mathit{lastI}(p)\},$$

where

$$\mathit{firstI}(1) = 1, \qquad \forall_{p=1}^{P-1} : \mathit{lastI}(p) + 1 = \mathit{firstI}(p+1), \qquad \text{and} \qquad \mathit{lastI}(P) = N.$$

This range is determined in an initial call to a load balancing routine, of which gathering the corresponding portion of *JNB*, namely *myJNB*, is the most time consuming part.

Each processor stores the data for *all* atoms:

$$\mathit{Stored}(p) = \{I \mid 1 \leq I \leq N\}.$$

According to the local memory model employed by ℙFORTRAN, there is no implicit consistency enforcement for these data across processors. This avoids unnecessary communication, but necessitates an explicit accumulation of the forces at the end of the computation, as shown in the code in Figure 13. Assuming an effective load balancer, this implementation results in

$$T_{comp} \approx \mathcal{O}(Pairs_{ave}) \qquad \text{and} \qquad T_{comm} \approx \mathcal{O}(N),$$

where the latter bound is achieved by performing the accumulation step with $2 \log P$ messages per processor using a divide-and-conquer approach [FJL+88].

A FORTRAN D version of this kind of algorithm can be written by expanding each array by one dimension (the *processor dimension*), the introduced index being the processor number, and then distributing that dimension blockwise [CHK+92].

### A.3.2 The data locality of UHGromos

The global atom numbering provided by sequential GROMOS and the resulting pair list do not inherently have a good locality; *i.e.*, atoms close together in space do not necessarily have similar *gan*'s. Even in sequential GROMOS, this is relevant for the cache efficiency, similarly for UHGROMOS. However, it becomes even more important when we want to increase the scalability of UHGROMOS (where the maximal problem size is limited by having all data replicated) by *distributing* the data across processors. To analyze this locality quantitatively, we ran the SOD test case already mentioned in Section A.2.2 on four processors. Figure 14 shows the resulting values for *AllWrites*, *OwnedWrites*, and *NotOwnedWrites*. Note that load balancing resulted in having $|\mathit{Owned}(p)|$ increase as $p$ increases, which is a consequence of Equation 1 again. This also caused that no other processor wrote to the atoms owned by processor 1; as this graph indicates, however, most (if not all) other atoms receive many contributions by processors other than their owner. Given the $f_{global}$ of current machines (Section 2.2), distributing the data according to their ownership (*i.e.*, setting $\forall_p : \mathit{Stored}(p) = \mathit{Owned}(p)$) and furthermore issuing one message for each non-local data access would therefore have a devastating effect on performance.

However, one might hope to improve the situation by *message blocking*, *i.e.*, collecting all non-local contributions any processor has until the end of the computation and then performing the communication. This would imply that

- All non-local reads have to be performed *before* the computation, using for example an inspector-executor approach as described in Section 2.3, and
- All non-local contributions of $p$ have to be buffered in $p$'s local memory, in addition to those owned by $p$,

but it might still decrease overall communication costs and/or increase scalability. How beneficial message blocking really is depends on how the non-local data accesses are distributed across processors:

- If each processor would contribute only to a *few atoms* other than his own, we would need less buffer space, which would improve scalability.
- If for each processor the atoms it makes non-local contributions to would be owned by a *few processors*, then $T_{comp}$ would go down.

To characterize UHGROMOS under this aspect, we have to break *AllWrites*, which summarizes across processors, up into *Writes*, which reflects the access patterns of individual processors. Figure 15 shows the resulting data for processor 2 when running the SOD test case with four processors. We observe that processor 2 makes some non-local contributes to all atoms owned by processors 3 and 4 (ignoring that the applied smoothing factor might hide a few atoms that were not hit by processor 2). Since we already assume that we perform message blocking, it does not help that the number of non-local contributions *per atom* is lower than the number of local contributions; what matters is whether for a particular atom there are any contributions at all or not.

One can conclude that – at least in this particular test case – there is little to be gained by distributing the data, and there is no evidence that other test cases result in better locality. Furthermore, it appears that even for higher numbers of processors each processor still contributes to most other processors unless $P \gg INB_{ave}$ [CHMS92].

### A.3.3 The UHGromos approach from the compiler's point of view

To summarize this approach:

1. *Replicate* data across processors: can be done in FORTRAN D as indicated in Section A.3.1.
2. *Reduce loop bounds* for compute intensive loops:
   (a) identify this loop,
   (b) need a load balancing metric for appropriately reducing loop bounds.
3. *Insert global reduction operation.*

Evaluation:

- Apart from the load balancing, which would require a relatively complex interface, this seems feasible.
- However, scalability is limited.

## A.4 Value-Based Decompositions A — The EulerGromos Approach

EULERGROMOS uses a more involved decomposition scheme that retains the spatial coherence of the simulated system. The Eulerian decomposition offers the advantage of providing locality for all parts of GROMOS with the potential for scalability [EHL77, HT84, PT91b]. In the remainder of this section, we will discuss the principal data structures of EULERGROMOS.

### A.4.1 Logical Eulerian space

We conceptually divide our overall *problem domain*, which here is the physical space occupied by the set of atoms we want to simulate, into small rectilinear regions of fixed size, henceforth called *subboxes*. We map the application space into a logical coordinate system of subboxes. Each subbox conceptually contains a list representation of the atoms resident within its spatial extent [EHL77]. Connected sets of subboxes are assigned to processors according to some space-to-processor mapping strategy. Each of these connected sets constitutes a *subdomain* assigned to a certain processor. Each subdomain $s$ is associated with a certain *overlap area*, which is the set of subboxes that are not in $s$ but reach into the cutoff radius of some subbox in $s$.

In our implementation, the data structure on a processor $p$ for a subbox $s$ consists of just a pointer to the head of a linked list of the atoms stored on $p$ whose $X$-coordinates are within $s$. Each processor $p$ has a copy of a structure representing the full problem domain, with non-empty pointers for each subbox that contains atoms stored on $p$. It would be possible to restrict the subbox structure to just the part that contains local atoms. This could further increase the scalability of our code, but it would make it more difficult to do dynamic load balancing involving the shift of processor subdomains.

Current mapping and load balancing strategies include *blockwise*, *slicewise*, and *hierarchical* decompositions (see Section 3.2.1). The first two techniques are straightforward decompositions of the problem domain into equal sized rectangular boxes or slices. For illustrating the hierarchical decomposition, let us assume we have $P = p_1 p_2 p_3$ processors, with $p_d$ processors for dimension $d$. The hierarchical decomposition first cuts the problem domain into $p_1$ slices, then it divides each of these slices into $p_2$ strips, and finally it cuts each strip into $p_3$ subdomains. Were these subdomains of fixed, identical size, the result would be the same as for the blockwise decomposition. However, the hierarchical decomposition also enables us to create subdomains of different sizes; for a balanced workload, the subdomains should be larger if they correspond to a region with lower density. This can be achieved by choosing the cuts along the three dimensions in such a manner that each resulting slice/strip/subdomain contains a roughly equal number of atoms. This is illustrated in the example shown in Figure 16.

For each spatial dimension $d$, the number of subboxes, $n_d$, and their size, $box_d$, depend on several parameters including the number of processors $P$, the mapping strategy, the number of atoms $N$, and the cutoff radius $R_c$. There are several tradeoffs and constraints to be observed:

- We distribute our problem domain across processors with *subbox granularity*, *i.e.*, a certain subbox is treated as indivisible as far as ownership goes, and we assume only one owner per subbox.

  Therefore, if $n_d$ becomes smaller, our load balancing may become less accurate, since the number of different decompositions becomes smaller.

- However, if $n_d$ becomes larger, the overhead associated with a traversal of the subboxes to locate the atoms increases.

- We also use our subbox structure to limit our search for nonbonded interaction partners of a given atom, which enables us to avoid the naïve $\mathcal{O}(N^2)$ pairlist generation algorithm. For that purpose it is advantageous if $box_d$ is an exact fraction of $R_c$ [PT91b].

- We expect that for molecular systems with *varying* density $\rho$, the hierarchical decomposition can outperform the blockwise and slicewise decompositions by adjusting subdomain sizes dynamically. However, for the trivial case of a system with *constant* $\rho$, the hierarchical decomposition should also be able to balance the workload. Therefore it must be possible to create subdomains of equal size; so for all $d$, $n_d$ should be a multiple of $p_d$.

### A.4.2 The nonbonded force calculation in EulerGromos

The nonbonded force subroutine required only slight changes from standard GROMOS. The original control structure of the nested loop shown in Figure 11 was modified to the code in Figure 17. Note that $X$, $V$, and $F$ are actually three-dimensional values.

The only conceptual difference is in how the local atoms are accessed. EULERGROMOS has two ways of looping over local atoms. The first method is to iterate directly over local data arrays, like in Figure 17, where an outer loop iterates from 1 to $N_{local}$, the number of local atoms. The second technique loops over data indirectly using the subbox data structure described in Section A.4.1. In both cases, the loop complexity is on the order of the number of atoms local to the processor, and *not* the total number of atoms in the system; this property is important for scalability.

In a molecular-dynamics simulation, molecules undergo a wide variety of motion that includes localized, high-frequency vibrations of bonds, diffusion of solvent molecules, and large-scale motion of protein subunits. As a result, the local atom set of a processor changes as a simulation evolves. To accommodate this shifting of atoms between processors, EULERGROMOS maintains two views of the data, a *local view* and a *global view*.

Each processor maintains a local naming convention for both the local atoms and the buffered neighbors. To maintain scalability, local data structures, such as the coordinate array, are accessed using a *lan* (local atom number). Thus a processor uses $\mathcal{O}(N/P)$ space for coordinates, velocities and forces. Exchanged atoms, however, are identified using the *gan* (global atom number) as introduced in Section A.2. The gan is used to directly access various atom-dependent parameters and actually conforms to the canonical GROMOS view of the molecular system.

By translating gan's to lan's for relocated atoms, processors avoid the shipping of parameters associated with a lan-identified atom. Having a uniform, global atom number for each atom gives us easy access to atom type, atom charge, and topology information. Without such a unifying scheme, we would need to piggy-back each atom changing ownership with this additional information.

The gan-to-lan translation is done via a mapping array *GAN2LAN*, which is the only local $\mathcal{O}(N)$ data structure besides the molecular-topology data structures; this trade of memory for speed is feasible for most practical applications [CHMS92]. However, should the simulation size exceed the processing-element memory's capability for storing this data structure, we could still distribute it (at the expense of additional communication) or replace it with a hash table (with higher overhead than simple array look-up and still potential for extra communication).

### A.4.3 The EulerGromos approach from the compiler's point of view

To summarize this approach:

1. Need to change data structures from arrays to linked lists connected by data structure reflecting spatial locality (*very* difficult to automate).
2. Use load balancer to distribute data accordingly.
3. Implement overlap regions for buffering atoms within cutoff.
4. Generate messages for updating overlap regions.

Advantages:

- Obviously we get good computational locality when cutting this data structure into $P$ simple, connected pieces. This locality is also used for the pairlist generation (*i.e.*, *no more* $\mathcal{O}(N^2)$*!*)
- Can use compact ownership *function*.

Disadvantages:

- If we still want to use a simple data structure (like $n$-dimensional arrays), then we have to provide storage for a certain maximum density. Otherwise, data structures can become really complicated.
- Important information, for example about the spatial topology, is based on the *global atom number* (*GAN*). This number is uniquely attached to each atom, and we often have to reference atoms by their GAN. This becomes difficult when atoms move around in a data structure according to their spatial movements.
- Obviously, the whole approach is relatively complicated. There are plenty of publications about just how which data structures can be used to effectively exploit this spatial locality, and Euler Gromos has already taken many more man hours to implement than UH Gromos.

Evaluation:

- Will probably give good performance results (this hope was the reason for starting with the EulerGromos project after all). Therefore, it is a useful baseline against which the performance of automated approaches can be measured.
- However, especially the conversion of the data structures can probably not be automized to a high degree; we consider it to be beyond current compiler technology.

## A.5 Value-Based Decompositions B — The CHARMM-ICASE Approach

This is work in progress, carried out by R. Das and J. Saltz at ICASE. The basic approach is to derive decompositions from the data access pattern as given by the nonbonded pairlist (see Section A.2).

1. Generate the pairlist as usual, *i.e.*, $\mathcal{O}(N^2)$ (sequentially) or $\mathcal{O}(N^2/P)$ (parallel).
2. Run a partitioner to distribute data according to the locality preferences given by the pairlist.
3. Compute an ownership array as defined in Section 3.2.1.
4. Use Parti routines to generate schedules and gathers/scatters.

Advantages:

- Conceptually relatively straightforward.
- Can use PARTI directly.

Disadvantages:

- The mapper can be expensive.
- Need (distributed) mapping *array*, two step process for determining ownership (but we still need some mechanism like this to keep track of GAN's anyway, as described in Section A.3.1).

## A.6 Value-Based Decompositions C — A Hybrid Approach

Let $X$ be the array where the positions of the atoms are stored. Conceptually, do the following:

1. Like in Approach B, leave data arrays, like $X$ (atom positions), $V$ (velocities), and $F$ (forces), as simple arrays (instead of more complicated data structures like in Approach A).
2. Initially, use some regular decomposition to store $X$.
3. Compute or read in initial atom positions and store them in $X$.
4. Apply a mapping *function* (like recursive bisection) to the *values* of $X$.
5. Based on this mapping, derive a mapping *array* for the *indices* of $X$. (However, the user should not see the mapping array at any point, it should only see the mapping function.)
6. Define a decomposition, like *AtomD*, and distribute it according to that mapping array.
7. Align $X$, $V$, and $F$ with *AtomD*.
8. Handle the resulting communication patterns with calls to PARTI routines.

The difference to Approach A is that we keep using simple arrays, where the indices do not necessarily have any spatial meaning. However, we do derive the decomposition from the spatial locations associated with the data points, instead of from the pairlist as in Approach B.

A (very preliminary) syntax for this kind of decomposition could look like the following:

```
DECOMPOSITION  AtomD(MaxAtom)
DISTRIBUTE     AtomD(BLOCK)
ALIGN          X, F, V  WITH  AtomD

... initialize X ...

DISTRIBUTE     AtomD  BY VALUE  X  USING  RecBis
```

Some points to be made:

- `RecBis` here stands for *recursive bisection*, which indicates the mapper to use. This mapper should also provide the possibility to provide some parameters, like the problem domain size or the metric for dividing space. However, if no such parameters are provided by the programmer, the mapper should be able to use default values and still produce reasonable results for standard problem configurations.
- This mapper could either be part of the language (just like BLOCK or CYCLIC), or it could be supplied as a library package compatible with a an interface provided by the compiler (*see Fox's proposals in this direction*).
- The second `DISTRIBUTE` should be viewed as an executable here.
- Remapping has to be done in some intervals. We do not expect the compiler to derive *when* this has to be done; this has to be indicated by the user, for example by executing

  ```
  DISTRIBUTE    AtomD  BY VALUE  X  USING  RecBis
  ```

  again.
- One might also try to provide some concept of overlap areas, but (at least initially) this might be handled implicitly by the PARTI routines.

Advantages:

- Probably faster mapper than Approach B.
- Simpler and closer to original program than Approach A.
- Fits nicely into FORTRAN D framework.
- Easy to keep track of GAN's.

Disadvantages:

- Still need mapping array like in Approach B (but again, we still need some mechanism like this to keep track of GAN's anyway, see Section A.3.1).

# B A Dataflow Framework for Optimizing Communication Placement

This appendix is organized as follows. Section B.1 provides some definitions and terminology for the framework. Section B.2 introduces the local flow variables, followed by global variables in Section B.3 and result variables in Section B.4. Section B.6 gives an extension of the framework for handling reduction operations. In Section B.7, we work through the dataflow variables for a program example, which is a simplified version of a mesh solver, which, together with experimental results based on the outcome of this framework, is described in more detail elsewhere [HKK+92].

## B.1 Basics of the Framework

### B.1.1 The domain

Even though our implementation can handle other cases as well, we assume here for presentation purposes that all indirect references in the program text are of the form $\langle array \rangle$ ($\langle index\_array \rangle$ ($\langle loop\_variable \rangle$)). For many programs, this can actually be achieved by forward substituting array indices. For example, the code sequence `j=ia(i); x(j)=10` would be treated as `x(ia(i))=10`. Arrays that are never referenced indirectly are assumed to be analyzed using other methods [GS90] prior to this analysis. References with multiple (but bounded) levels of indirection will require more levels of complexity in the dataflow framework; we do not consider potentially unbounded indirection, as is found in linked lists.

Let $V$ be the set of arrays that are accessed indirectly; in the example code shown in Figure 5 it is $V = \{x, y, z\}$. We assume that each reference $r$ to some $v \in V$ is contained in some loop(s). Let $L$ be the set of loops that *directly* enclose an occurrence of some $v \in V$; in the example, it is $L = \{l_1, l_2, l_3, l_4\}$. We assume that no $l \in L$ encloses any other $m \in L$. One set of dataflow variables is computed for each element of a set of *nodes*, $N$. It is $N = L \cup P$, where $P$ contains one *entry pad*, $l_{entry}$, and one *exit pad*, $l_{exit}$, for each loop $l \notin L$ containing some $l' \in L$. The example contains one outer time stepping loop, for which we introduce $l_{entry} = l_0$ and $l_{exit} = l_5$. This results in $P = \{l_0, l_5\}$ and $N = \{l_0, l_1, l_2, l_3, l_4, l_5\}$. Furthermore, we assume $l_{entry}$ ($l_{exit}$) to be executed before (after) $l$ iff $l$ has *at least one iteration*; this enables us to hoist communication out of loops without risking unnecessary communication in case the loop has zero iterations. In the example, the resulting loop flow graph shown in Figure 7 therefore has an edge around the outer loop *including* $l_0$ and $l_5$.

The framework operates on a *loop flow graph* $G = (N, E)$ of the program, where the edges $E$ are simple control flow edges. For example, if $l$ is an outer time stepping loop that does not directly contain any irregular array references but contains a loop $l'$ over mesh edges, then $l'$ is represented as a node in $G$ and $l$ is represented as some interval in $G$. In the following, *loop* refers to elements of $N$, *i.e.*, it may denote a pad as well.

Future work will present a complete framework in which summary information is built in a bottom-up fashion similar to array kill information [GS90]. Finally, we only discuss the case where the summarized loops have no data dependences, except for commutative and associative reductions that are handled specially.

### B.1.2 Array portions

Array portions are a central concept to the framework and best introduced by an example. A *portion* `x(ia(1:n))` consists of the *array* `x` and the *index set* `ia(1:n)`. This index set in turn consists of the *index array* `ia` and the *range* `(1:n)`, which has the *lower bound* `1` and the *upper bound* `n`.

Several portions may be taken from the same array or may have the same index set. The index range does not have to be known at compile time, so the bounds may contain symbolics. No assumptions are made about whether different portions taken from the same array are disjoint or whether they overlap each other partially or completely. This enables analyzing symbolic index ranges, but it requires the analysis to be conservative when using intersection and set subtraction in the equations.

The framework can be implemented using a lattice of bit vectors. Each bit vector represents a dataflow variable at a certain node in $G$, and each bit represents one array portion. To construct these bit vectors, an initial pass over the program has to collect all indirect references. The length of the bit vectors is bounded by the number of indirect array references an therefore linear in program size. All equations given here are *rapid* [KU76]. Therefore, using bit vectors for the analysis gives us good asymptotic running times. However, for our examples (and probably also in a practical implementation), it seems advantageous to represent the different flow variables as *bit matrices*. The rows of a bit matrix correspond to the arrays of the portions represented (*e.g.*, `x` in `x(ia(1:n))`), while the columns correspond to the index sets (`ia(1:n)`). Theoretically that representation increases variable sizes from linear in program size to quadratic in program size, so the feasibility of this approach depends on how programs behave in practice. However, this representation makes potential schedule sharing, for example, very easy to recognize by determining which index set columns have more than one entry.

We assume that all indirect array references are identified in a previous pass over the program text and construct bit vectors/matrices accordingly. For the analysis we also assume that a (identity) dummy index array is inserted for all direct array references.

### B.1.3 Operations on portions

To aid the distinction between portions, indirect array references, array elements, and sets of all these constructs, we make a short digression to introduce the conversion operators *elements-of* $p$ (where $p$ is some portion or set of portions), denoted $\tilde{p}$, and *references-of* $p$, denoted $\tilde{\tilde{p}}$. Assume we are given

- an array $x$;
- an index array $ia(1:5)$;
- portions $p = x(ia(p_l : p_u))$, $q = x(ia(q_l : q_u))$, $r = x(ia(r_l : r_u))$; and
- sets of portions $A = \{p, q\}$, $B = \{p\}$, $C = \{r\}$.

We can reason about $A$, $B$, and $C$ at different *levels*.For example, if the index ranges of the portions are only known symbolically, one can determine at the *portion level* that

$$A \supseteq B$$

must hold, but no other relationships can be proven among the sets of portions. However, if we know for example that

$$p_l = 1, p_u = 3, q_l = 3, q_u = 5, r_l = 3, r_u = 4,$$

then the elements-of operator, $\tilde{}$, can be applied to the portions and to the sets thereof, to obtain

$$\tilde{A} = x(ia(1:5)), \tilde{B} = x(ia(1:3)), \tilde{C} = x(ia(3:4)).$$

The scope of $\tilde{}$ is extended to set operators and predicates, so we can assert at the *element level* that

$$A \tilde{\supseteq} B, \qquad\qquad A \tilde{\supseteq} C.$$

Assume furthermore that we know the values of the index array to be

$$ia(1:5) = 1, 4, 3, 1, 4.$$

Then the references-of operator, $\tilde{\tilde{}}$, obtains

$$\tilde{\tilde{A}} = \{x(1), x(3), x(4)\}, \tilde{\tilde{B}} = \{x(1), x(3), x(4)\}, \tilde{\tilde{C}} = \{x(1), x(3)\}.$$

With this knowledge, we conclude at the *reference level* that

$$A \tilde{\tilde{\supseteq}} B \tilde{\tilde{\supseteq}} C.$$

We can see how the set relationship predicates change over the different levels of reasoning, with

$$X \supseteq Y \Longrightarrow X \tilde{\supseteq} Y \Longrightarrow X \tilde{\tilde{\supseteq}} Y.$$

Another interesting operation in this context is set subtraction:

- $A \setminus B = \{p, q\} \setminus \{p\} = \{q\}$, which is $\{x(1), x(3), x(4)\}$;
- $A \tilde{\setminus} B = \tilde{A} \setminus \tilde{B} = x(ia(1:5)) \setminus x(ia(1:3)) = x(ia(4:5))$, which is $x(1), x(4)\}$;
- $A \tilde{\tilde{\setminus}} B = \tilde{\tilde{A}} \setminus \tilde{\tilde{B}} = \{x(1), x(3), x(4)\} \setminus \{x(1), x(3), x(4)\} = \emptyset$.

As described in Section B.4, $A \setminus B$ (and the corresponding sets at lower levels) can be viewed as a so called incremental schedule, which indicates what has to be communicated if $A$ is needed and $B$ is already available in local memory. We can see immediately the consequences for this incremental schedule in the example: the more we know about portions, the less we might have to communicate. Formally,

$$X \setminus Y \;\supseteq\; X \tilde{\setminus} Y \;\supseteq\; X \tilde{\tilde{\setminus}} Y.$$

To aid formulating conservative equations that still offer the possibility to exploit any knowledge potentially available at compile time, we introduce some set operators that map sets of portions into sets of portions. Given some set of portions $SET$, we define

$$\begin{aligned}
SET^{*} &= \{p \mid p \text{ has same } \textit{array} \text{ as some } q \in SET\}, \\
SET^{\cup} &= \{p \mid SET \text{ might } \textit{affect } p\} \\
&= \{p \mid p \tilde{\tilde{\cap}} SET \neq \emptyset \text{ cannot be disproven}\} \\
&\subseteq SET^{*}, \\
SET^{\cap} &= \{p \mid SET \textit{ contains } p\} \\
&= \{p \mid p \tilde{\tilde{\subseteq}} SET \text{ can be proven}\} \\
&\supseteq SET, \\
SET^{\circ} &= \{p \mid SET \text{ might } \textit{partially touch} \text{ part of } p\} \\
&= SET^{\cup} \setminus SET^{\cap}.
\end{aligned}$$

$SET^{*}$ can be derived easily from $SET$ by just reducing a bit matrix (array names by index sets) to a bit column (array names) using row-wise OR. From there we can conservatively approximate $SET^{\cup}$, $SET^{\cap}$, and $SET^{\circ}$ directly, or we can employ further compile time knowledge about how portions relate to each other if available. Either way, we do not leave the portion space as given in the program, *i.e.*, we can still represent these sets with binary bit matrices.

For example, let the portions $p$, $q$, $r$ be defined as above, and let $D = \{q\}$. If no compile time knowledge at the element or reference level is available, then we conservatively assume that $D^{*} = \{p, q, r\}$, $D^{\cup} = \{p, q, r\}$, $D^{\cap} = \{q\}$, and $D^{\circ} = \{p, r\}$. With knowledge at the element level, we have $D^{*} = \{p, q, r\}$, $D^{\cup} = \{p, q, r\}$, $D^{\cap} =$

$\{q, r\}$, and $D^{\circ} = \{p\}$. Reference level knowledge gives $D^{*} = \{p, q, r\}$, $D^{\cup} = \{p, q, r\}$, $D^{\cap} = \{p, q, r\}$, and $D^{\circ} = \emptyset$.

A point to keep in mind when reasoning about which elements are contained in which portions and how portions relate to each other is that two portions $p$, $q$ might globally contain the same set of array elements of some array $X$, but that *locally* a given processor may see different parts of $X$ for $p$ and $q$. (This applies to lhs occurrences as well, since we apply the *owner computes rule* based on index array ownerships, *not* on data array ownerships; otherwise we would not need a SCATTER operation). In this case communication must occur if for example we first define $p$ and then use $q$. The important consequence is that we must apply $\tilde{}$ and $\tilde{\tilde{}}$ based on the share of each processor.

Furthermore, we have to keep different decompositions of arrays and index arrays in mind for the analysis. For example, we cannot reuse a schedule between two portions that have the same index set, but whose arrays are distributed differently. For sake of simplicity, however, we assume in this paper that all arrays are conformable.

## B.2 The Local Flow Variables

We define the *local flow variables* to be the components of the dataflow equations that are determined by local analysis of each loop. In the following,

- $l$ stands for an arbitrary loop node,
- $p$ denotes a portion `x(ia(lb:ub))`,
- an *occurrence* of $p$ is either a use of $p$ or a definition of $p$, and
- the terms "variable" or "flow variable" stand for dataflow variables.

We begin with two variables, *REF* and *DEF*, which are familiar from standard live variable analysis. A point to keep in mind, however, is that here *live* does not refer to whole arrays, but to limited portions thereof instead. Also, there may be conditionals in the loops generating the variables, which can be handled by annotating portions with (symbolic) guards applying to whole portions or elements thereof.

For each loop $l$, we define

**REF(l):** the portions *live* on entry to $l$, and

**DEF(l):** the portions *defined* in $l$.

Formally:

$$\begin{aligned} REF(l) &= \{p \mid \textit{first} \text{ stmt containing } p \text{ in } l \text{ reads } p\}, \\ DEF(l) &= \{p \mid \text{some stmt in } l \text{ assigns to } p\}. \end{aligned}$$

To aid the extension to reduction statements discussed in Section B.6, we do not base the further development of the framework on *REF* and *DEF* directly, but replace them with *GET* and *PUT*. These variables are used to derive the portions that have to be buffered locally. We define

**GET(l):** the portions referenced in $l$ from local memory (the *buffer*).

**PUT(l):** the portions written by $l$ into the buffer.

**BUF(l):** the portions that will be buffered on exit from $l$.

The equations (which will be redefined in Section B.6):

$$\begin{aligned} GET(l) &= REF(l), \\ PUT(l) &= DEF(l), \\ BUF(l) &= GET(l) \cup PUT(l). \end{aligned}$$

We also have to compute the live ranges of index sets, otherwise we might accidentally try to communicate a portion before or after the program region where its index set is available (*i.e.*, before the index set is defined or after it is overwritten with other values). We define

**IND(l):** the portions whose index sets may be computed (in part) by $l$.

**KILL(l):** the portions that may be made invalid by $l$, either because $l$ assigns an overlapping part of the array or $l$ reassigns the index set. GATHER operations can never be hoisted above $l$ for these portions.

**FLUSH(l):** the portions that may be read by $l$ or whose index sets may be reassigned by $l$. SCATTER operations can never be delayed until after $l$ for these portions.

Formally:

$$\begin{aligned} IND(l) &= \{p \mid p \text{ has index set } \texttt{ia(i}_{min}\texttt{:i}_{max}\texttt{)} \text{ and } l \text{ assigns to } \texttt{ia}\}, \\ KILL(l) &= IND(l) \cup DEF^{\circ}(l), \\ FLUSH(l) &= IND(l) \cup REF^{\circ}(l). \end{aligned}$$

## B.3 The Global Flow Variables

The computation of the *global flow variables* constitutes the meat of the dataflow framework. Here we actually propagate knowledge about the communication characteristics of the loops around in the flow graph. The problems addressed here have elements from Common Subexpression Elimination, Loop Invariant Code Motion, and Dead Code Elimination. As already mentioned, all equations given here are *rapid*, so we can expect to solve them efficiently using simple iterative techniques. All global variables are initialized to $\emptyset$.

### B.3.1 Fetches

The strategy for determining where to place GATHER operations is based on the following definitions:

**LIVE$^{\mathbf{any/all}}$(l):** the portions that are needed in $l$ or along any/all paths starting in $l$.

**BUFFD(l):** the portions that are already available when entering $l$. Here we assume that buffers are not flushed unless the data in them may be invalid, because either the data array or the index array has been assigned to.

**HOIST(l):** the portions for which a GATHER should be hoisted ahead of $l$.

**FETCH(l):** the portions that are needed in $l$, or needed in some later loop and can be hoisted before $l$.

The equations:

$$\begin{aligned}
\mathit{LIVE}^{all}(l) &= \mathit{GET}(l) \cup \bigcap_{s \in succs(l)} (\mathit{LIVE}^{all}(s) \setminus \mathit{KILL}(l)),\\
\mathit{LIVE}^{any}(l) &= \mathit{GET}(l) \cup \bigcup_{s \in succs(l)} (\mathit{LIVE}^{any}(s) \setminus \mathit{KILL}(l)),\\
\mathit{BUFFD}(l) &= \mathit{BUF}(l) \cup \bigcap_{p \in preds(l)} (\mathit{BUFFD}(p) \setminus \mathit{KILL}(l)),\\
\mathit{HOIST}(l) &= \bigcap_{p \in preds(l)} (\mathit{LIVE}^{all}(p) \cup \mathit{BUFFD}(p)),\\
\mathit{FETCH}(l) &= \mathit{GET}(l) \cup \bigcap_{s \in succs(l)} (\mathit{HOIST}(s) \cap \mathit{FETCH}(s)).
\end{aligned}$$

At this point, we have identified candidate locations in the program for placing GATHER's. In short, whenever a portion appears in a *FETCH*(l) set, then that portion can be gathered before $l$ and will be used before it is assigned. The final placement will be determined by the result flow variables discussed in Section B.4.

Note that we can not only distinguish the variables defined so far by whether they are local or global, but we can also classify them into either reflecting fixed properties *inherent* of the analyzed program, or being subject to *heuristics*. Furthermore, this classification can be done based either on the *definition* of the variable, *i.e.*, how it is defined in terms of other variables, or on the actual *values* of the variable.

For example, *HOIST* is currently defined so that we *combine and hoist up GATHER's as much as possible*, subject to the constraint that we never want to overcommunicate. If we, for example, replace the $\mathit{LIVE}^{all}$ in the definition of *HOIST* with $\mathit{LIVE}^{any}$, we could hoist up communication even further, at the expense of possibly communicating unnecessary data, but with the potential benefit of additional schedule saving. Another strategy would be to limit communication hoisting to cases where we actually reduce the size or number of messages, which can also be achieved in a straightforward manner similar to *lazy code motion* [KRS92]; this might decrease the live ranges of our communication buffer with a possible savings in overall buffer storage requirements, but at the expense of reduced opportunities for hiding communication delays.

In other words, the *definition* of *HOIST* is a matter of *heuristics*, which is not the case for the other definitions so far. For other variables dependent on *HOIST* (so far, *FETCH* is the only such variable), their *values* become a matter of the chosen heuristics as well, but not their definition.

### B.3.2 Stores

The high level strategy for determining where to place SCATTER operations is relatively similar to the one for placing GATHER's. Note that we do not have to scatter portions (*i.e.*, send them back to the owner) if they are used only locally, which is why we restrict our attention to $\mathit{GET}^{\circ}$ instead of *GET*. The definitions:

**HIN$^{\mathbf{any/all}}$(l) / HOUT$^{\mathbf{any/all}}$(l):** the portions touched by a reference on any/all of the paths starting at the entry/exit of $l$.

**DELAY(l):** the portions that should be scattered in a later loop, or are dead on exit.

**STORE(l):** portions that are assigned to in $l$, or were assigned to earlier and whose SCATTER's can be hoisted into $l$.

$$\begin{aligned}
HIN^{all}(l) &= GET^{\circ}(l) \cup HOUT^{all}(l), \\
HOUT^{all}(l) &= \bigcap_{s \in succs(l)} HIN^{all}(s), \\
HIN^{any}(l) &= GET^{\circ}(l) \cup HOUT^{any}(s), \\
HOUT^{any}(l) &= \bigcup_{s \in succs(l)} HIN^{any}(l), \\
DELAY(l) &= \bigcap_{s \in succs(l)} (HOUT^{all}(s) \cup \overline{HOUT^{any}(s)}) \setminus \bigcup_{s \in succs(l)} FLUSH(s), \\
STORE(l) &= PUT(l) \cup \bigcap_{p \in preds(l)} (DELAY(p) \cap STORE(p)).
\end{aligned}$$

Our heuristic, here defined by *DELAY*, is to combine and delay SCATTER's as much as possible, subject to the constraint that we never scatter data that are dead.

## B.4 The Result Flow Variables

The *result flow variables* given in this section are computed after solving the equations given so far. They should accurately describe which portions have to be gathered before entering $l$ or scattered after leaving $l$ (possibly using reductions). Here we want to take previous and succeeding loops and their communication requirements into account as well.

### B.4.1 Fetches

Similarly to *FETCH*, *GATH*(l) describes which portions have to be in local memory before entering $l$. However, it excludes portions that must already be locally available either by previous gathers or by previous calculations. Furthermore, we may not only exclude these available data on a portion by portion basis, but also on an element by element basis. In other words, if we know that a portion `x(ia(i`$_{min}$`:i`$_{max}$`))` is buffered, then we might not only eliminate gathers of exactly that portion, but we can also save on a gather of a potentially overlapping portion `x(ia(j`$_{min}$`:j`$_{max}$`))` by gathering only the *increment* from the first portion to the second one.

For that purpose we compute *incremental schedules* using the $\tilde{\tilde{\setminus}}$ operator as introduced in Section B.1; recall that $A \tilde{\tilde{\setminus}} B$ contains exactly those references that appear in the portions in $A$ but do not appear in any of the portions in $B$. Note that this operator, unlike the $\setminus, \cup, \cap$ used in the flow equations so far, brings us out of the fixed space of sets of portions appearing in the program text, and applying it repeatedly can lead to an explosion of the number portions we have to be able to represent (nestings of increments of intersections of increments, etc.). Applying this operator just once, however, leads to sets that can still be represented by 3-valued "bit" vectors/matrices; in addition to *included/not included*, we also need *explicitly excluded.*

Note also that $A \tilde{\tilde{\setminus}} B = \emptyset$ is possible even for $A \setminus B \neq \emptyset$. This reflects for example the case where we express a mesh and its boundary as different portions of the same array; the portions are distinct, but one contains a subset of the other.

The equation:

$$GATH(l) \;=\; FETCH(l) \mathbin{\tilde{\tilde{\setminus}}} \bigcap_{p \in preds(l)} (FETCH(p) \cup BUFFD(p)).$$

### B.4.2 Stores

The *SCATT* variables are derived from the *STORE* variables, except that we eliminate unnecessary scatters by excluding portions that either will be scattered later, or are not at least potentially live (using $\overline{HOUT^{any}}$). Again, we use the set operator $\tilde{\tilde{\setminus}}$ to support incremental schedules.

$$SCATT(l) \;=\; STORE(l) \mathbin{\tilde{\tilde{\setminus}}} \bigcap_{s \in succs(l)} (STORE(s) \cup \overline{HOUT^{any}(s)}).$$

Note that we can still override the communication patterns obtained by global analysis for *GATH* and *SCATT* by just substituting the local counterparts *GET* and *PUT* for them, as long as this is done consistently for all loops. Furthermore, this can be done for either both variables or for just one of them, since they do **not** rely on each other, but merely on the loop properties.

## B.5 Schedules

The framework described so far gives an accurate description of which schedules are needed where. Critical for the overall cost associated with our communications is also the *generation* of these schedules, in particular *where* the schedules are generated. However, once we know the communication requirements, schedule computation placement appears to be relatively straightforward. Therefore, we currently use the simple heuristic of generating schedules as soon as possible, *i.e.*, as soon as the necessary index arrays are available. This seems to work well in the codes we have considered so far.

## B.6 Reduction Variables

As indicated earlier, the framework developed so far can be extended to take advantage of reduction statements as well. The portions exclusively appearing in reduction statements can be treated differently from other definitions and uses, since they are not necessarily brought into local memory if we use reduction operations like SCATTERADD or SCATTERMULT. However, portions appearing in different reduction operations within one loop have to be brought into local memory, so we have to carefully separate the portions into the ones used exclusively in *ADD* reductions and the ones used only in *MULT* reductions:

$$\begin{aligned} ADD(l) &= \{p \mid \text{all } q \in p^{\cup} \text{ are only added to in } l\}, \\ MULT(l) &= \{p \mid \text{all } q \in p^{\cup} \text{ are only multiplied to in } l\}. \end{aligned}$$

We derive *RED*, the set of all portions that are used exclusively in reduction operations, and redefine *GET* and *PUT* which were introduced in Section B.2:

$$\begin{aligned} RED(l) &= ADD(l) \cup MULT(l), \\ GET(l) &= REF(l) \setminus RED(l), \\ PUT(l) &= DEF(l) \setminus RED(l). \end{aligned}$$

The changes so far have eliminated the GATHER's and SCATTER's for portions that appear exclusively in reductions.

We now define another, separate framework, which computes *only* the SCATTER_ADD's (similarly for the other reductions). This new *ADD framework* coexists with the old *non-reduction framework*, which is still used to compute communication requirements for non-reduction operations. The redefined variables are:

$$\begin{aligned}
GET_{ADD}(l) &= REF(l) \setminus ADD(l), \\
FLUSH_{ADD}(l) &= IND(l) \cup GET^{\circ}{}_{ADD}(l), \\
STORE_{ADD}(l) &= ADD(l) \cup \bigcap_{p \in preds(l)} (DELAY_{ADD}(p) \cap STORE_{ADD}(p)).
\end{aligned}$$

Corresponding to these new variables, we can derive $HIN^{any/all}_{ADD}$, $HOUT^{any/all}_{ADD}$, $DELAY_{ADD}$, and $SCATT_{ADD}$ with the same equations as for the non-reduction framework. $SCATT_{ADD}$ now indicates where to place SCATTER_ADD's.

In the exact same fashion we can define a *MULT framework* by substituting *ADD* with *MULT*. Like for the non-reduction framework, we can override the result variable with their local counterpart, which is here *ADD* (*MULT* in the MULT framework). Note that the flow equations for ADD (MULT) are defined independently of other reductions. This simplifies extending the framework to other reduction operations by just adding flow variables and equations, without having to modify existing ones (except extending *RED*).

## B.7 Example

Figure 5 shows an example code, which is a simplified and abstracted version of an actual unstructured mesh solver [DMS+92].

The loop structure of the example code can be derived from the actual solver by inlining the function calls; the main loop in the example is analogous to a time-stepping loop.

In this program, we have

- four inner loops, $l_1$, $l_2$, $l_3$, and $l_4$;
- three array names, $x$, $y$, and $z$;
- five index sets, $s_1 = ie1(1:ne)$, $s_2 = ie2(1:ne)$, $s_3 = if1(1:nf)$, $s_4 = if2(1:nf)$, and $s_5 = identity(1:nn)$;
- this spans a bit matrix of fifteen portions, $x_1 = x(s_1), x_2 = x(s_2), \ldots, z_5 = z(s_5)$, twelve of which actually occur in the program text.

The corresponding flow graph is shown in Figure 7; note the entry pad $l_0$ and the exit pad $l_5$ of the outer time stepping loop, which does not directly enclose any indirect references, but which contains other loops ($l_1, \ldots, l_4$) that contain such references.

The bit matrices of the resulting local flow variables are shown in Figure 18. A matrix entry for a particular portion $p$ and a flow variable *VAR* is defined as follows:

**"1"** – $p$ is *included* in *VAR*,

**"_"** – $p$ is *not included* in *VAR*,

**"0"** – $p$ is *explicitly excluded* from *VAR* (as a result of the $\tilde{\tilde{\setminus}}$ operator; in our example, there are none such entries due to the simple control flow structure).

Figure 19 shows the global and result variables, and Figure 20 shows the variables for the ADD framework. The loop flow graph including the result flow variables is shown in Figure 8.

The result variables, *i.e.* *GATH* and *SCATT*, determine where the GATHER and SCATTER operations should be placed. If the bit representing a portion $p$ is set in the *GATH* set, then a GATHER operation for $p$ is placed at the *beginning* of that loop. Similarly, a set bit in the *SCATT* set results in placement of a SCATTER operation (SCATTER_ADD in the ADD framework) at the *end* of a loop. GATHER's and SCATTER's of portions with *identity* as the index array are ignored because they represent data movement from a processor to itself. The resulting code is shown in Figure 6.

We do not show here the optimizations needed to generate the *schedule* operations (*i.e.*, the inspectors). In general, the method is to identify the index sets used, and insert the inspectors at the birth points of those sets. The first step can be done by inspection, while the second is a simple application of reaching definition analysis.

```
  DO t = 1, itime

C    Loop l_1
     DO i = 1, ne
       x(ie1(i)) = x(ie1(i)) + y(ie2(i))
       x(ie2(i)) = x(ie2(i)) + y(ie1(i))
     ENDDO

C    Loop l_2
     DO j = 1, nf
       x(if1(j)) = x(if1(j)) + y(if2(j)) + z(if2(j))
       x(if2(j)) = x(if2(j)) + y(if1(j)) + z(if1(j))
     ENDDO

C    Loop l_3
     DO k = 1, ne
       x(ie1(k)) = x(ie1(k)) + y(ie2(k))
       x(ie2(k)) = x(ie2(k)) + y(ie1(k))
     ENDDO

C    Loop l_4
     DO l = 1, nn
       y(l) = x(l)
     ENDDO

  ENDDO
```

**Figure 5** Original example code, using a global name space.

```
IF (itime ≥ 1)
  GATHER(z(if1(1:nf)), z(if2(1:nf)))
  DO t = 1, itime

C    Loop l_1
     GATHER(y(ie1(1:ne)), y(ie2(1:ne)),
          y(if1(1:nf)), y(if2(1:nf)))
     DO i = 1, ne_loc
       x(ie1(i)) = x(ie1(i)) + y(ie2(i))
       x(ie2(i)) = x(ie2(i)) + y(ie1(i))
     ENDDO

C    Loop l_2
     DO j = 1, nf_loc
       x(if1(j)) = x(if1(j)) + y(if2(j)) + z(if2(j))
       x(if2(j)) = x(if2(j)) + y(if1(j)) + z(if1(j))
     ENDDO

C    Loop l_3
     DO k = 1, ne_loc
       x(ie1(k)) = x(ie1(k)) + y(ie2(k))
       x(ie2(k)) = x(ie2(k)) + y(ie1(k))
     ENDDO
     SCATTERADD(x(ie1(1:ne)), x(ie2(1:ne)),
              x(if1(1:nf)), x(if2(1:nf)))

C    Loop l_4
     DO l = 1, nn_loc
       y(l) = x(l)
     ENDDO

  ENDDO
ENDIF
```

**Figure 6** Parallelized example code, using a local name space. (The schedule generation and most of the parameters for the communication calls are omitted for clarity.)

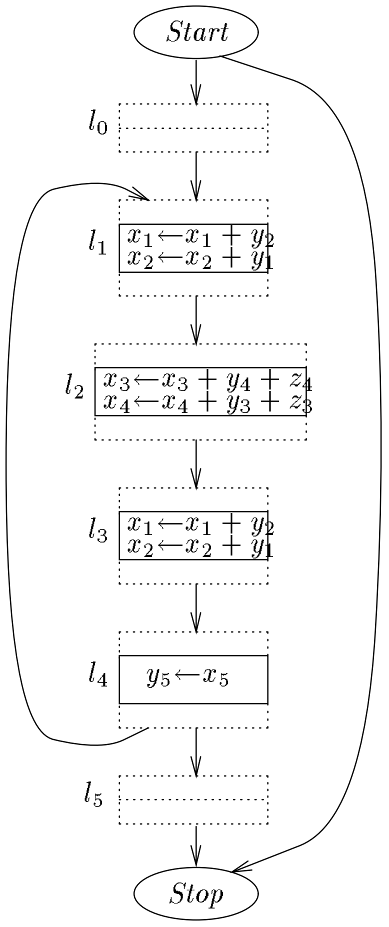
Start
$l_0$
$l_1$
$x_1 \leftarrow x_1 + y_2$
$x_2 \leftarrow x_2 + y_1$
$l_2$
$x_3 \leftarrow x_3 + y_4 + z_4$
$x_4 \leftarrow x_4 + y_3 + z_3$
$l_3$
$x_1 \leftarrow x_1 + y_2$
$x_2 \leftarrow x_2 + y_1$
$l_4$
$y_5 \leftarrow x_5$
$l_5$
Stop

**Figure 7** Flow graph for example code. The dotted rectangles indicate potential call sites for PARTI routines to gather data (at beginning of loop) and scatter data (at end of loop).

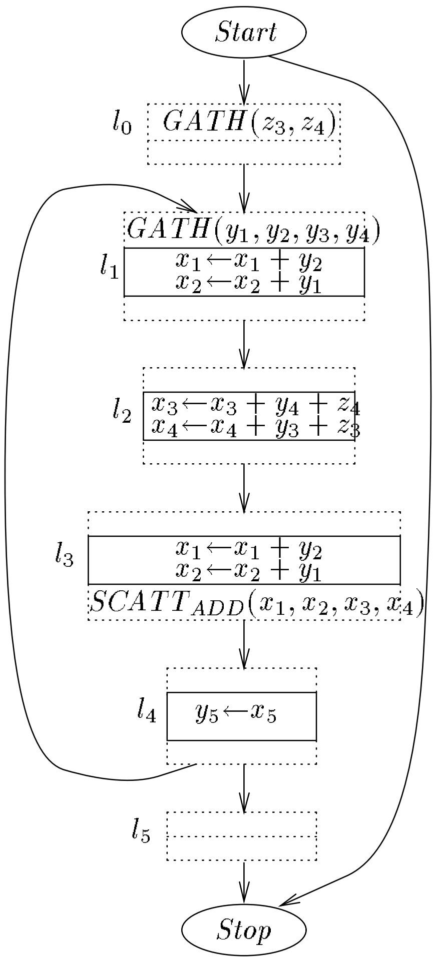
Start
$l_0$
$GATH(z_3, z_4)$
$GATH(y_1, y_2, y_3, y_4)$
$l_1$
$x_1 \leftarrow x_1 + y_2$
$x_2 \leftarrow x_2 + y_1$
$l_2$
$x_3 \leftarrow x_3 + y_4 + z_4$
$x_4 \leftarrow x_4 + y_3 + z_3$
$l_3$
$x_1 \leftarrow x_1 + y_2$
$x_2 \leftarrow x_2 + y_1$
$SCATT_{ADD}(x_1, x_2, x_3, x_4)$
$l_4$
$y_5 \leftarrow x_5$
$l_5$
Stop

**Figure 8** Flow graph for example code, with calls to PARTI routines.

```
init_1                          i = 1
WHILE test_1                    WHILE (i ≤ K)
  init_2                          j = 1
  WHILE test_2                    WHILE (j ≤ L(i))
    BODY                            X(i,j) = i * j
    increment_2                     j = j + 1
  ENDWHILE                        ENDWHILE
  increment_1                     i = i + 1
ENDWHILE                        ENDWHILE
```

**Figure 9** Generic loop nest (left) and corresponding EXAMPLE (right).

```
$\text{init}_1$
$\text{init}_2$
WHILE $\text{test}_1$
  $\mathcal{BODY}$
  IF $\text{done}_2$ THEN
    $\text{increment}_1$
    $\text{init}_2$
  ELSE
    $\text{increment}_2$
  ENDIF
ENDWHILE
```

```
i = 1
j = 1
WHILE $(i \le K)$
  X(i,j) = $i * j$
  IF (j = L(i))
    i = i + 1
    j = 1
  ELSE
    j = j + 1
  ENDIF
ENDWHILE
```

**Figure 10** Flattened generic loop nest and `EXAMPLE`.

```
DO I = 1, N
  DO Jind = firstJ(I), lastJ(I)
    J = JNB(Jind)
    force = nbf(X(I) − X(J))
    F(I) = F(I) + force
    F(J) = F(J) − force
  ENDDO
ENDDO
```

**Figure 11** Sequential form of the nonbonded force calculation.

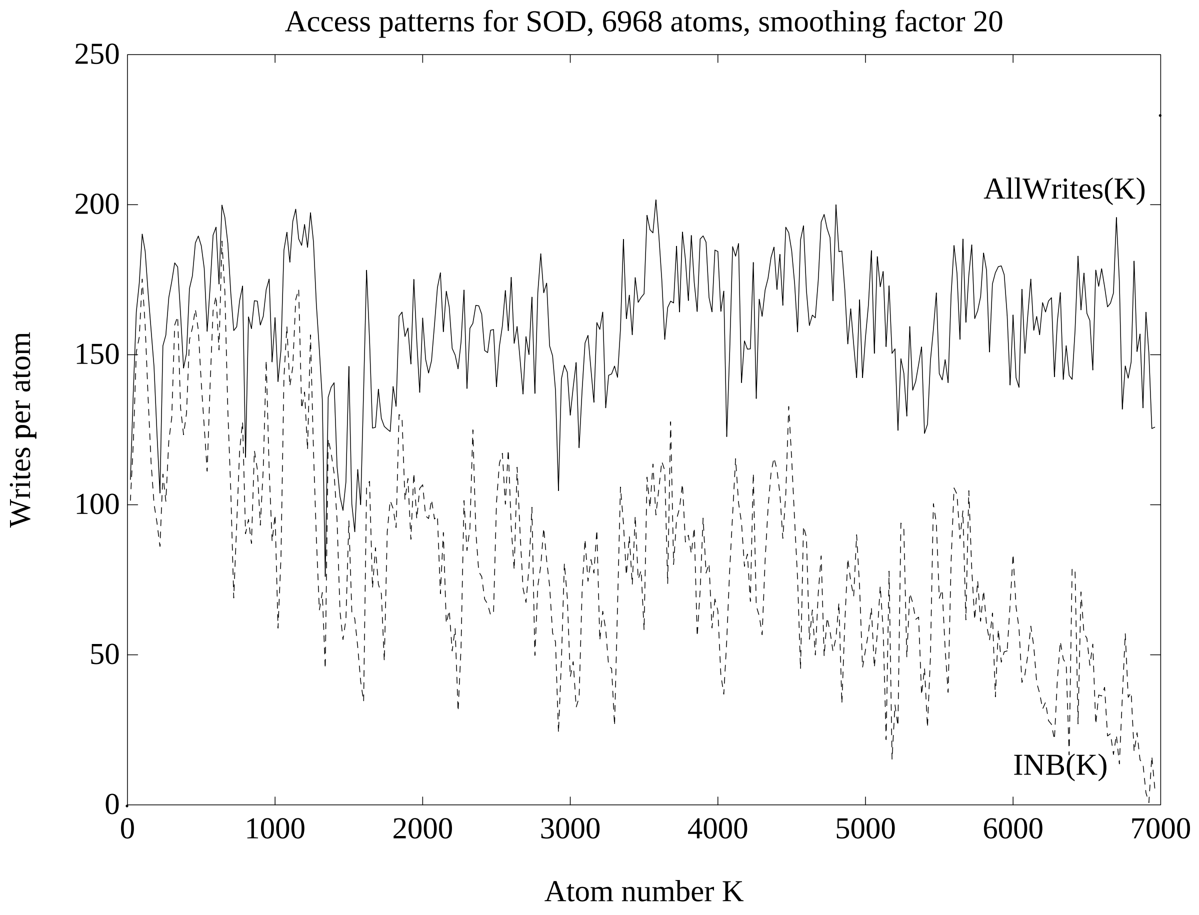

**Figure 12** The total number of *Writes* to each atom (solid) and the contributions by *INB* (dashed). It is

```
Balance(firstI, lastI)

DO I = firstI(me), lastI(me)
   DO Jind = firstJ(I), lastJ(I)
      J = myJNB(Jind)
      force = nbf(X(I) − X(J))
      F(I) = F(I) + force
      F(J) = F(J) − force
   ENDDO
ENDDO

F = +{F}
```

**Figure 13** Force calculation, UHGROMOS version.

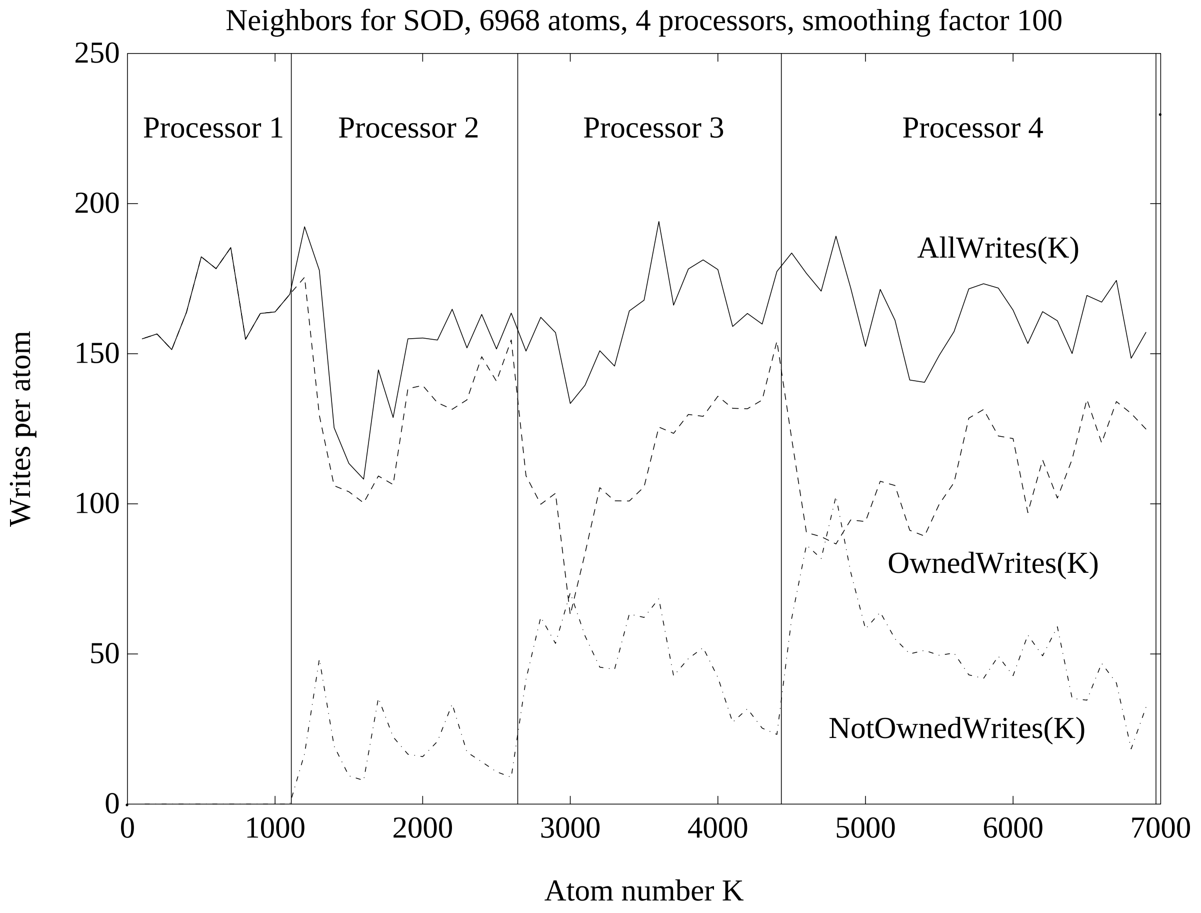

**Figure 14** The total number of *Writes* to each atom (solid), broken down into local contributions *OwnedWrites* (dashed) and nonlocal contributions *NotOwnedWrites* (dot-dashed).

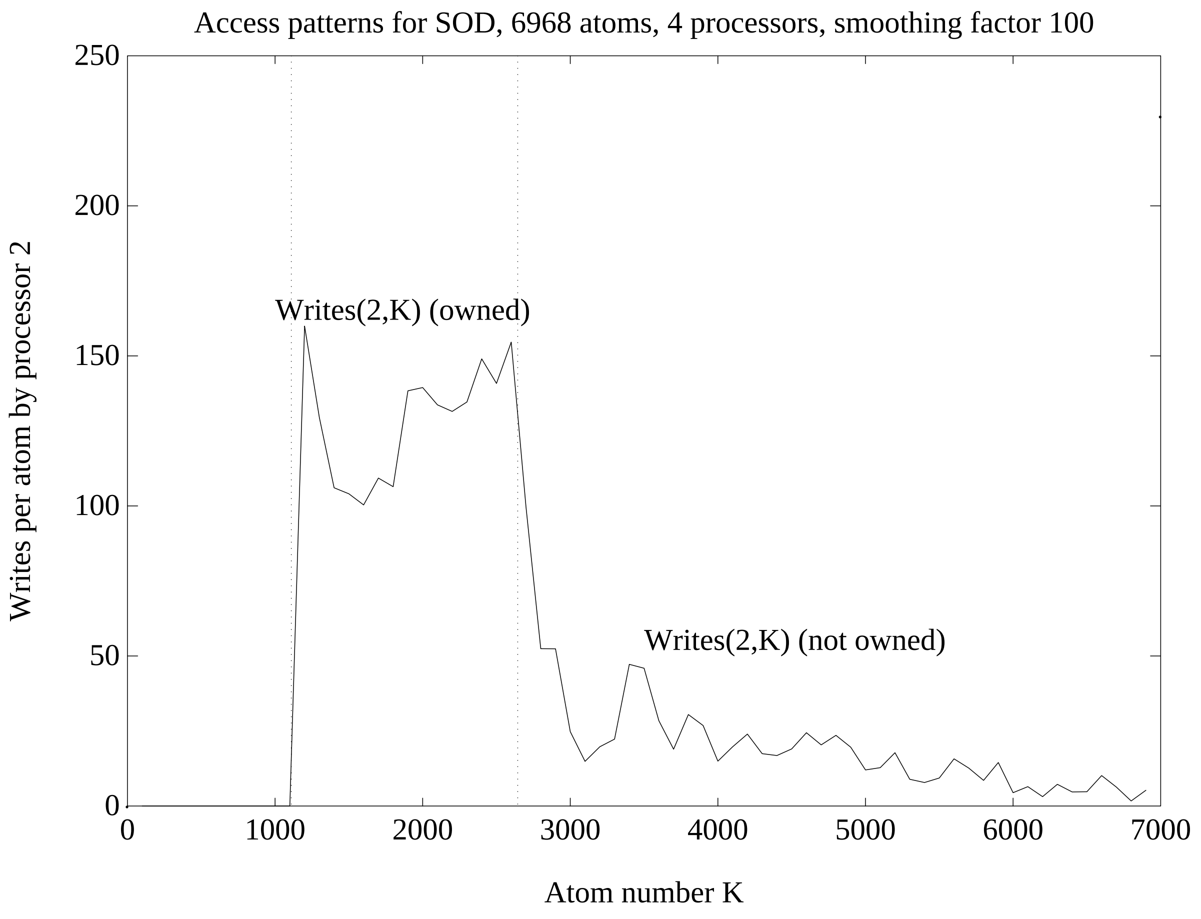

**Figure 15** The contributions of processor 2 to each atom, $Writes(2, K)$. The vertical dotted lines indicate the ownership range of processer 2.

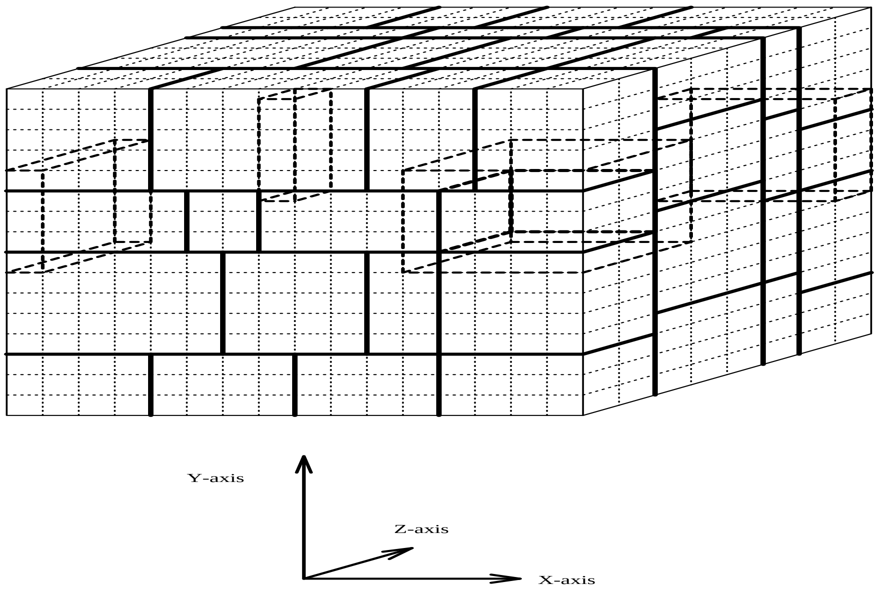

**Figure 16** Subbox division of the problem domain; we use $P = p_1p_2p_3 = 4*4*4$ processors and $n_{tot} = n_1n_2n_3 = 16*16*8$ subboxes. Dotted lines indicate subboxes, heavy lines delineate processor subdomains. Hashed-lined regions show the overlap area of the processor with logical coordinate $(4,3,1)$, which has a subdomain consisting of $4*3*2$ subboxes located at the center of the edge closest to the reader; note the wrap-around of the overlap due to periodic boundary conditions.

```
DO I = 1, $N_{local}$
  DO Jind = firstJ(I), lastJ(I)
    J = JNB(Jind)
    force = nbf(lan2gan(I),lan2gan(J))
    F(I) = F(I) + force
    IF (is−local(J)) THEN
      F(J) = F(J) − force
    ENDIF
  ENDDO
ENDDO
```

**Figure 17** EulerGromos version of the nonbonded force calculation.

|  | $l_0$ | $l_1$ | $l_2$ | $l_3$ | $l_4$ | $l_5$ |
|---|---|---|---|---|---|---|
|  | _____ | 11___ | __11_ | 11___ | ____1 | _____ |
| *REF* | _____ | 11___ | __11_ | 11___ | _____ | _____ |
|  | _____ | _____ | __11_ | _____ | _____ | _____ |
|  | _____ | 11___ | __11_ | 11___ | _____ | _____ |
| *DEF* | _____ | _____ | _____ | _____ | ____1 | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | 11___ | __11_ | 11___ | _____ | _____ |
| *ADD* | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | 11___ | __11_ | 11___ | _____ | _____ |
| *RED* | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | ____1 | _____ |
| *GET* | _____ | 11___ | __11_ | 11___ | _____ | _____ |
|  | _____ | _____ | __11_ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
| *PUT* | _____ | _____ | _____ | _____ | ____1 | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | ____1 | _____ |
| *BUF* | _____ | 11___ | __11_ | 11___ | ____1 | _____ |
|  | _____ | _____ | __11_ | _____ | _____ | _____ |
|  | _____ | __111 | 11__1 | __111 | _____ | _____ |
| *KILL* | _____ | _____ | _____ | _____ | 1111_ | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | __111 | 11__1 | __111 | 1111_ | _____ |
| *FLUSH* | _____ | __111 | 11__1 | __111 | _____ | _____ |
|  | _____ | _____ | 11__1 | _____ | _____ | _____ |

**Figure 18** Local flow variables for example code.

|  | $l_0$ | $l_1$ | $l_2$ | $l_3$ | $l_4$ | $l_5$ |
|---|---|---|---|---|---|---|
|  | _____ | _____ | _____ | _____ | ____1 | _____ |
| $LIVE^{all}$ | 1111_ | 1111_ | 1111_ | 11___ | _____ | _____ |
| $\Longleftarrow$ | __11_ | __11_ | __11_ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | ____1 | _____ |
| $LIVE^{any}$ | 1111_ | 1111_ | 1111_ | 11___ | _____ | _____ |
| $\Longleftarrow$ | __11_ | __11_ | __11_ | __11_ | __11_ | _____ |
|  | _____ | _____ | _____ | _____ | ____1 | ____1 |
| $BUFFD$ | _____ | 11___ | 1111_ | 1111_ | ____1 | ____1 |
| $\Longrightarrow$ | _____ | _____ | __11_ | __11_ | __11_ | __11_ |
|  | _____ | _____ | _____ | _____ | _____ | ____1 |
| $HOIST$ | _____ | _____ | 1111_ | 1111_ | 1111_ | ____1 |
| $\longrightarrow$ | _____ | __11_ | __11_ | __11_ | __11_ | __11_ |
|  | _____ | _____ | _____ | _____ | ____1 | _____ |
| $FETCH$ | _____ | 1111_ | 1111_ | 11___ | _____ | _____ |
| $\Longleftarrow$ | __11_ | __11_ | __11_ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | ____1 | _____ |
| $GATH$ | _____ | 1111_ | _____ | _____ | _____ | _____ |
| $\longrightarrow$ | __11_ | _____ | _____ | _____ | _____ | _____ |
|  | 1111_ | 1111_ | 1111_ | 1111_ | _____ | _____ |
| $HOUT^{all}$ | 11111 | 11111 | __111 | _____ | _____ | _____ |
| $\Longleftarrow$ | 11__1 | 11__1 | _____ | _____ | _____ | _____ |
|  | 1111_ | 1111_ | 1111_ | 1111_ | 1111_ | _____ |
| $HOUT^{any}$ | 11111 | 11111 | 11111 | 11111 | 11111 | _____ |
| $\Longleftarrow$ | 11__1 | 11__1 | 11__1 | 11__1 | 11__1 | _____ |
|  | 11___ | __11_ | 11___ | ____1 | 11___ | 11111 |
| $DELAY$ | 11___ | __11_ | _____ | _____ | 11___ | 11111 |
| $\longleftarrow$ | 11111 | __11_ | __11_ | __11_ | 11111 | 11111 |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
| $STORE$ | _____ | _____ | _____ | _____ | ____1 | _____ |
| $\Longrightarrow$ | _____ | _____ | _____ | _____ | _____ | _____ |
|  | _____ | _____ | _____ | _____ | _____ | _____ |
| $SCATT$ | _____ | _____ | _____ | _____ | ____1 | _____ |
| $\longleftarrow$ | _____ | _____ | _____ | _____ | _____ | _____ |

**Figure 19** Global flow variables and result flow variables for example code.

|  | $l_0$ | $l_1$ | $l_2$ | $l_3$ | $l_4$ | $l_5$ |
|---|---|---|---|---|---|---|
| $GET_{ADD}$ | _____<br>_____<br>_____ | _____<br>11___<br>_____ | _____<br>__11_<br>__11_ | _____<br>11___<br>_____ | ____1<br>_____<br>_____ | _____<br>_____<br>_____ |
| $FLUSH_{ADD}$ | _____<br>_____<br>_____ | _____<br>__111<br>_____ | _____<br>11__1<br>11__1 | _____<br>__111<br>_____ | 1111_<br>_____<br>_____ | _____<br>_____<br>_____ |
| $HOUT^{all}_{ADD}$ $\Longleftarrow$ | 1111_<br>11111<br>11__1 | 1111_<br>11111<br>11__1 | 1111_<br>__111<br>_____ | 1111_<br>_____<br>_____ | _____<br>_____<br>_____ | _____<br>_____<br>_____ |
| $HOUT^{any}_{ADD}$ $\Longleftarrow$ | 1111_<br>11111<br>11__1 | 1111_<br>11111<br>11__1 | 1111_<br>11111<br>11__1 | 1111_<br>11111<br>11__1 | 1111_<br>11111<br>11__1 | _____<br>_____<br>_____ |
| $DELAY_{ADD}$ $\longleftarrow$ | 11111<br>11___<br>11111 | 11111<br>__11_<br>__11_ | 11111<br>_____<br>__11_ | ____1<br>_____<br>__11_ | 11___<br>11___<br>11111 | 11111<br>11111<br>11111 |
| $STORE_{ADD}$ $\Longrightarrow$ | _____<br>_____<br>_____ | 11___<br>_____<br>_____ | 1111_<br>_____<br>_____ | 1111_<br>_____<br>_____ | _____<br>_____<br>_____ | _____<br>_____<br>_____ |
| $SCATT_{ADD}$ | _____<br>_____<br>_____ | _____<br>_____<br>_____ | _____<br>_____<br>_____ | 1111_<br>_____<br>_____ | _____<br>_____<br>_____ | _____<br>_____<br>_____ |

**Figure 20** Flow variables for ADD framework of example code.

# C Loop Flattening

This Section is organized as follows. Section C.1 describes the different variants of pseudo Fortran used in the examples. Section C.2 presents a small example to illustrate the kind of problem we are interested in and gives a first glance at loop flattening, which Section C.3 elaborates at at a more general level. Section C.4 examines the applicability of loop flattening for the nonbonded force kernel described in Section A.2, which we implemented on both the CM2 and the DECmpp; performance results on this can be found elsewhere [HK92]. Section C.5 evaluates loop flattening from the compiler perspective.

## C.1 Languages

The concepts introduced here apply to a broad range of languages. We will give program examples in different variants of pseudo Fortran:

**F77** - Strictly sequential Fortran 77 (possibly a "dusty deck" program).

**F77D** - F77 enhanced with decomposition statements as proposed in FORTRAN D [FHK+90] and High Performance Fortran [Hig92]. An important goal of F77D is to provide a basis for efficient compilation towards both MIMD and SIMD distributed memory machines, so it should not contain any constructs that are specific to either architecture.

**F77**$_{MIMD}$ - A Fortran 77 version to run on a MIMD machine, which assumes a separate name space for each processor.

**F90**$_{SIMD}$ - A Fortran 90 version to run on a SIMD machine, similar to Connection Machine Fortran [Thi91] or MasPar Fortran [Mas91]. There are two important differences to the F77 variants:

- By default, scalars of the F77 version will be *replicated* in the F90$_{SIMD}$ version; *i.e.*, they will be declared as vectors of size $P$, where processor $p$ owns the $p$-th element.
- In keeping with Fortran 90 convention, omitted array indices refer to all elements of an array dimension, and an unsubscripted array reference refers to all array elements.

For enhancing readability of the F90$_{SIMD}$ examples, we extend the language constructs that are typically implemented by vendors in several ways:

- The FORALL construct cannot only be applied to single statements, but also to blocks. The general form of this extension can be interpreted differently depending on the semantics chosen for the case where different iterations modify the same set of data; our examples, however, will avoid these access interferences.
- DO-ENDDO's, DO-WHILE's, IF's, WHERE's, and FORALL's can be nested freely within each other.
- WHILE loops can be controlled by an array of booleans (instead of just a scalar boolean), if the different array elements are guaranteed to have identical values.

```
C     P1 – sequential version
      DO i = 1, K
        DO j = 1, L(i)
          X(i,j) = i * j
        ENDDO
      ENDDO
```

**Figure 21** Original loop nest `EXAMPLE`.

## C.2 Example of Loop Flattening

Consider the contrived F77 loop nest in Figure 21, henceforth called `EXAMPLE`. This clearly is a dependence free, parallelizable loop, where the number of inner loop iterations depends on the current iteration of the outer loop. Let $K$ be 8 and let $L(1{:}8)$ have the values 4,1,2,1,1,3,1,3, respectively. Assuming $P = 2$ processors and the *owner computes rule*, where in all assignment statements the right hand side expression is computed by the processor that "owns" the left hand side variable, we can in this case just distribute $L$ and the rows of $X$ blockwise to achieve perfect load balance. This is illustrated in the F77D program in Figure 22, which assigns $L(1{:}4)$, $X(1{:}4{,}1{:}4)$ to the first processor and $L(5{:}8)$, $X(5{:}8{,}1{:}4)$ to the second processor. The owner computes rule results in partitioning the iteration space among the two processors, so each processor executes only some iterations of the outer loop.

For a MIMD machine, the Fortran D compiler would derive the $\mathrm{F77}_{MIMD}$ program shown in Figure 23. Each processor executes the loop nest independently, needing a total of

$$TIME_{MIMD} = \max_{p=1,2} \sum_{i=1}^{4} L(i + 4(p-1)) = 8 \qquad (2)$$

inner loop iterations. This is illustrated in the trace in Figure 24.

A $\mathrm{F90}_{SIMD}$ version could be derived from the F77D program by just changing the outer DO loop to a FORALL loop. This would result in a partitioning of the iteration space, similar to the F77D version. For expository reasons, we will give a slightly different but equivalent $\mathrm{F90}_{SIMD}$ version that takes the data decomposition and the number of processors already into account and thus directly reflects the control flow for $K = 8$ and $P = 2$. As in the $\mathrm{F77}_{MIMD}$ version, we change the upper bound of the outer loop from $K = 8$ to $K/P = 4$ and let each processor execute all iterations of the loop. We continue to use the loop index $i$ in control flow related statements; to enable the different processors to operate on different data, we introduce an auxiliary induction variable, $i'$, which replaces $i$ in non-control flow statements. The result is shown in Figure 25.

Note how we had to transform the inner DO loop due to the single SIMD control flow. To make sure that each processor can perform all of its iterations, the upper bound $L(i')$ had to be changed into the maximum of $L(i')$ over all processors. This in turn necessitated a guard for the loop body that tests whether this processor is still

```
C     P2 – Fortran D version
      DECOMPOSITION XD(K,Lmax), LD(K)
      ALIGN X with XD, L with LD
      DISTRIBUTE XD(BLOCK,*), LD(BLOCK)

      DO i = 1, K
        DO j = 1, L(i)
          X(i,j) = i * j
        ENDDO
      ENDDO
```

**Figure 22** `EXAMPLE` in F77D.

```
C     P3 – MIMD version
      DO i = 1, 4
        DO j = 1, L'(i)
          X'(i,j) = i * j
        ENDDO
      ENDDO
```

**Figure 23** `EXAMPLE` in F77$_{MIMD}$. $X$ and $L$ are renamed to $X'$ and $L'$ to reflect that there is no common name space any more. On processor $p$, $p = 1, 2$, $L'(i)$ corresponds to $L(i + 4(p - 1))$, and $X'(i, j)$ corresponds to $X(i + 4(p - 1), j)$.

involved in the current inner loop iteration or whether it is masked out and sits idle, possibly to participate again in later iterations.

We will refer to this transformation, which can be applied to other loop types as well, as *SIMDizing* a loop. It is a straightforward consequence of the SIMD restricted control flow, yet it is the crucial motivation for the concepts introduced in this paper. The outer loop does not have to be SIMDized in this particular case because we know that each processor works on exactly four rows of $X$ and therefore has to execute the outer loop the same number of times. Loop SIMDizing has the effect that our F90$_{SIMD}$ program has to execute

$$TIME_{SIMD} = \sum_{i=1}^{4} \max_{p=1,2} L(i + 4(p - 1)) = 12 \tag{3}$$

iterations. Roughly speaking, our time bound has increased from a maximum over sums to a sum over maxima. This becomes apparent when considering the execution trace shown in Figure 26.

Since the equivalent MIMD implementation performs significantly better, this bad running time can not be explained with lack of parallelism or bad load balance. To

| Time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $i_1$ | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 4 |
| $j_1$ | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 1 |
| $i_2$ | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 |
| $j_2$ | 1 | 1 | 2 | 3 | 1 | 1 | 2 | 3 |

**Figure 24** MIMD execution trace for `EXAMPLE` loop, $i_p$ and $j_p$ denote $i$ and $j$ on processor $p$.

```
C     P4 – naive SIMD version
      DO  i = 1, 4
        i' = i + [0,4]
        DO  j = 1, max(L(i'))
          WHERE  (j ≤ L(i'))  X(i',j) = i' * j
        ENDDO
      ENDDO
```

**Figure 25** `EXAMPLE` in F90$_{SIMD}$. [0,4] denotes the two-element vector containing 0 and 4.

overcome this purely control flow related problem, we apply *loop flattening*, which will be introduced at a more general level in the next section. The result is shown in Figure 27. Now we can achieve the same time bound as in the MIMD implementation, needing only eight steps as shown in the trace in Figure 24.

The reader might have noticed that the loop body shown in Figure 27 is now always executed at least once for each outer loop iteration, which is equivalent to assuming $L(i) \geq 1$ for all $i$. Even though this is correct in our example, a more general loop flattening does not rely on this assumption, as we will see in the next section.

## C.3 General Loop Flattening

Assume that we are given two fully parallelizable nested loops like in the previous section; an extension of the following to deeper loop nests is straightforward. Each of the loops might be structured as a WHILE loop, a DO-WHILE loop, a simple DO or FORALL loop, or it might use conditional GOTO's. The transformation described here can be done either at the F77/ F77D level or at the F90$_{SIMD}$ level. For simplicity and generality, we will present it here on the F77 level. A corresponding F90$_{SIMD}$ version can always be directly derived by SIMDizing loops and replacing IF's with WHERE's.

As a first step, we *normalize* both loops by breaking their control pattern into three *phases* for each nesting level $l$; an initialization phase $init_l$, a guard $test_l$, and an incrementing step $increment_l$. For example, a control pattern like `DO var = lo, hi,`

| Time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i_1$ | 1 | 1 | 1 | 1 | 2 | | | 3 | 3 | 4 | | |
| $j_1$ | 1 | 2 | 3 | 4 | 1 | | | 1 | 2 | 1 | | |
| $i_2$ | 1 | | | | 2 | 2 | 2 | 3 | | 4 | 4 | 4 |
| $j_2$ | 1 | | | | 1 | 2 | 3 | 1 | | 1 | 2 | 3 |

**Figure 26** Execution trace for unflattened example loop; $i_p$, $j_p$ denote the actual iteration counts of processor $p$, no entry means "idle."

```
C     P5 – flattened SIMD version
      i = [1,5]
      K = [4,8]
      j  = 1
      WHILE  ANY(i ≤ K)
        WHERE  (i ≤ K)
          X(i,j) = i ∗ j
          WHERE  (j = L(i))
            i  = i + 1
            j  = 1
          ELSEWHERE
            j = j + 1
          ENDWHERE
        ENDWHERE
      ENDWHILE
```

**Figure 27** `EXAMPLE` in flattened F90$_{SIMD}$.

`stride` would be broken into $init_l \equiv$ `var = lo`, $test_l \equiv$ `(var` $\leq$ `hi)`, and $increment_l \equiv$ `var = var + stride`. The resulting loop nest `GENNEST` is shown in Figure 28, along with the corresponding version of the `EXAMPLE` from the previous section (of course, we usually expect $\mathcal{BODY}$ to contain more computational work than in `EXAMPLE`).

Since `GENNEST` conservatively tests for loop completion before entering the loop body, all loops can be brought into this normal form. To estimate the running time of the above code on $P$ processors, for processor $p$ let $K_p$ be the number of outer loop iterations and $L_p^i$ be the number of inner loop iterations for the $i$-th outer loop iteration. A straightforward MIMD version would then finish after

$$TIME_{MIMD} = \max_{p=1\ldots P} \sum_{i=1}^{K_p} L_p^i \qquad (1')$$

iterations.

A F90$_{SIMD}$ version could be derived by SIMDizing both WHILE loops and would

```
init_1                          i = 1
WHILE test_1                    WHILE (i ≤ K)
  init_2                          j = 1
  WHILE test_2                    WHILE (j ≤ L(i))
    BODY                            X(i,j) = i * j
    increment_2                     j = j + 1
  ENDWHILE                        ENDWHILE
  increment_1                     i = i + 1
ENDWHILE                        ENDWHILE
```

**Figure 28** Generic loop nest `GENNEST` (left) and corresponding `EXAMPLE` (right); original version after normalization.

execute

$$TIME_{SIMD} = \sum_{i=1}^{\max_{p=1}^{P} K_p} \max_{p=1\ldots P} L_p^i \qquad (2')$$

iterations. Again, if the number of iterations of the inner loop varies from one outer loop iteration to the next, then the restriction to a common program counter makes this SIMD implementation inefficient.

Since we do not know whether the evaluation of $test_l$ has any side effects, we introduce flags $t_l$ to store the results of evaluating the conditions $test_l$ before we make any other transformations, see Figure 29. So far, control flow is still unchanged.

The key idea of loop flattening is to make sure that each processor has a chance to advance to the next loop iteration where it participates in the execution of $\mathcal{BODY}$ before the control flow actually reaches $\mathcal{BODY}$. One requirement that follows immediately is that control variables (iteration counts etc.) are replicated to enable individual processors to advance independently to the next outer loop iteration whenever they are done with the current inner loop. Furthermore, we have to take $\mathcal{BODY}$ out of the part of the loop nest that handles the transition between different iterations of the inner and outer loop. Each processor should be able to execute $\mathcal{BODY}$ whenever it has still work left to do in this loop nest and the control flow reaches $\mathcal{BODY}$. In other words, $\mathcal{BODY}$ should be executed whenever $t_1$ is true, independent of $t_2$. The flattened loop version meeting these goals is shown in Figure 30.

As the reader might verify, we still execute exactly the same instructions in the same order and the same number of times as we did in the original loop nest. We also still have two nested loops. However, $\mathcal{BODY}$ is lifted out of the inner loop. The inner loop now contains just the control structure to let each processor advance to the next iteration in which it actually executes $\mathcal{BODY}$. In other words, *the processors still have to run through $\mathcal{BODY}$ and the rest of the loop nest in lockstep, but now they may be executing effectively different loop iterations.*

The above transformation is the most general, conservative one. It can be optimized for several special cases; one common case is that

1. $test_1$, $test_2$ and $init_2$ have no side effects, and that

```
init$_1$                                  i = 1
t$_1$ = test$_1$                          t$_1$ = (i $\leq$ K)
WHILE t$_1$                               WHILE t$_1$
 init$_2$                                   j = 1
 t$_2$ = test$_2$                           t$_2$ = (j $\leq$ L(i))
 WHILE t$_2$                                WHILE t$_2$
  $\mathcal{BODY}$                            X(i,j) = i $*$ j
  increment$_2$                               j = j + 1
  t$_2$ = test$_2$                            t$_2$ = (j $\leq$ L(i))
 ENDWHILE                                   ENDWHILE
 increment$_1$                              i = i + 1
 t$_1$ = test$_1$                           t$_1$ = (i $\leq$ K)
ENDWHILE                                  ENDWHILE
```

**Figure 29** GENNEST/EXAMPLE, with guard variables.

2. for each outer loop iteration, the inner loop is executed at least once.

Then we can safely transform the code into the simpler version shown in Figure 31.

If it further is the case that

3. we can replace the guard *test*$_2$ with a test whether we are in the last inner iteration, *done*$_2$ (for example, in `DO var = lo, hi, stride`, we can replace *test* $\equiv$ `(var` $\leq$ `hi)` with *done* $\equiv$ `(var = hi)`),

then we can save the last execution of *increment*$_2$, as shown in Figure 32. The SIMDized equivalent `EXAMPLE` of this version was shown in Figure 27.

## C.4 Case Study with Molecular Dynamics

The transformation described in the previous section should be profitable whenever some processors sit idle in an inner loop and still have work to do in later iterations of the outer loop. This seems to be a situation potentially occurring in many scientific programs solving irregular problems [BSGM90, SPBR91, TP90, WLR90]. One example is the GROMOS molecular dynamics program, which contains several interesting kernels of this kind [CHK$^+$92, CHMS92, GB88]. Here we want to focus on the calculation of the nonbonded forces between individual pairs of atoms which is described in Section A.2; Figure 11 shows the kernel of a typical sequential implementation.

Figure 33 shows a F90$_{SIMD}$ version that lays out the data in a cyclic fashion. If we assume for simplicity that $P$ divides $N$, then each processor computes the nonbonded forces for $N/P$ atoms. The uneven atom density results in varying values of *INB*; therefore, the inner loop with the (relatively expensive) force calculation often has to be executed with processors masked out even though they still have work to do in later iterations, just as it was the case in the `EXAMPLE` in Section C.2. All processors have

```
init$_1$                            i = 1
t$_1$ = test$_1$                    t$_1$ = (i $\leq$ K)
IF t$_1$ THEN init$_2$              IF t$_1$ then j = 1
WHILE t$_1$                         WHILE t$_1$
  t$_2$ = test$_2$                    t$_2$ = (j $\leq$ L(i))
  WHILE (t$_1$ $\wedge\neg$ t$_2$)    WHILE (t$_1$ $\wedge\neg$ t$_2$)
    increment$_1$                       i = i + 1
    t$_1$ = test$_1$                    t$_1$ = (i $\leq$ K)
    IF t$_1$ THEN                       IF t$_1$ THEN
      init$_2$                            j = 1
      t$_2$ = test$_2$                    t$_2$ = (j $\leq$ L(i))
    ENDIF                               ENDIF
  ENDWHILE                            ENDWHILE
  IF t$_1$ THEN                       IF t$_1$ THEN
    $\mathcal{BODY}$                    X(i,j) = i $*$ j
    increment$_2$                       j = j + 1
  ENDIF                               ENDIF
ENDWHILE                            ENDWHILE
```

**Figure 30** `GENNEST/EXAMPLE`, after flattening.

to go through

$$TIME_{SIMD} = \sum_{i=1}^{N/P} \max_{p=1\ldots P} INB(Atom_p^i) \qquad (2'')$$

iterations, where $Atom_p^i$ is the $i$-th Atom of processor $p$.

This can be improved on by applying loop flattening, where we take into account that each atom has at least one interaction partner. The result is shown in Figure 34. Now each processor can loop through its atoms individually, so this code achieves the same time bound as a MIMD implementation:

$$TIME_{SIMD}^{flat} = \max_{p=1\ldots P} \sum_{i=1}^{N/P} INB(Atom_p^i), \qquad (1'')$$

which is only limited by the quality of our workload distribution.

## C.5 Loop Flattening from the Compiler's Perspective

The discussion so far seems to advocate a certain style of SIMD programming for applications that can benefit from loop flattening, just as a certain style of programming emerged when vector machines became popular. However, this would be contrary to existing efforts to make programming independent from machine idiosyncrasies, as for example the development of the Fortran D language. For non-SIMD machines, it still seems natural and efficient to have the inner loop bodies contained in the inner

```
init$_1$                          i = 1
init$_2$                          j = 1
WHILE test$_1$                    WHILE (i $\leq$ K)
  $\mathcal{BODY}$                    X(i,j) = i $*$ j
  increment$_2$                     j = j + 1
  IF NOT test$_2$ THEN              IF NOT(j $\leq$ L(i))
    increment$_1$                     i = i + 1
    init$_2$                          j = 1
  ENDIF                             ENDIF
ENDWHILE                          ENDWHILE
```

**Figure 31** `GENNEST/EXAMPLE`, flattened and optimized.

```
init$_1$                          i = 1
init$_2$                          j = 1
WHILE test$_1$                    WHILE (i $\leq$ K)
  $\mathcal{BODY}$                    X(i,j) = i $*$ j
  IF done$_2$ THEN                  IF (j = L(i))
    increment$_1$                     i = i + 1
    init$_2$                          j = 1
  ELSE                              ELSE
    increment$_2$                     j = j + 1
  ENDIF                             ENDIF
ENDWHILE                          ENDWHILE
```

**Figure 32** `GENNEST/EXAMPLE` after further optimization.

loops, even though flattened loops should run well on these machines also. Therefore, we suggest to make loop flattening part of the optimizing repertoire of SIMD compilers.

*Applicability* is ensured whenever there are multiple loops fully contained in each other, *i.e.*, there are not several loops on the same nesting level. This can be easily derived from the abstract syntax tree. Furthermore, the normalized version always tests the loop guard $test_l$ before executing $\mathcal{BODY}$, so we cover all loop constructs. The transformation itself is relatively straightforward; for example, there are no parameters to adjust, unlike in loop skewing. The first step of the transformation is to identify the three phases *init*, *test*, and *increment*.

**WHILE/DO-WHILE loops:** The relevant phases can be identified from their position between the WHILE and ENDWHILE keywords. Since $increment_2$ and $\mathcal{BODY}$ stay together throughout the transformation, we actually do not need to separate these two phases.

**DO/FORALL loops:** The phases can be derived directly from the loop header, as exemplified earlier.

```
F = 0
I = [1 : P]
lastI = [N−P+1 : N]
WHILE ANY (I ≤ lastI)
  WHERE (I ≤ lastI)
    DO Jind = 1, max(INB(I))
      WHERE (Jind ≤ INB(I))
        J = JNBL(I, Jind)
        force = nbf(X(I) − X(J))
        F(I) = F(I) + force
        F(J) = F(J) − force
      ENDWHERE
    ENDDO
    I = I + P
  ENDWHERE
ENDWHILE
```

**Figure 33** F90$_{SIMD}$ version of `NBFORCE`.

```
F = 0
I = [1 : P]
lastI = [N−P+1 : N]
Jind = 1
WHILE ANY (I ≤ lastI)
  WHERE (I ≤ lastI)
    J = partners (I, Jind)
    force = nbf(X(I) − X(J))
    F(I) = F(I) + force
    F(J) = F(J) − force
    WHERE (Jind = INB(I))
      I = I + P
      Jind = 1
    ELSEWHERE
      Jind = Jind + 1
    ENDWHERE
  ENDWHERE
ENDWHILE
```

**Figure 34** Flattened F90$_{SIMD}$ version of `NBFORCE`. We take into account that $INB(i) \geq 1$ for all $i$.

**Reducible GOTO loops:** Similarly to WHILE loops, we can identify the phases by their position between labels and jumps.

After normalization, the introduction of flags $t_l$ and the actual code rearrangement follow straightforwardly. As described in Section C.3, we also can often detect opportunities for further optimizations, for example when we are transforming simple DO/FORALL loops.

In evaluating *profitability*, we note that the additional overhead caused by loop flattening is, in the worst case, to manipulate two flags and to perform two conditional jumps. So we can relatively safely assume profitability whenever the inner loop bounds may vary across the processors.

As with many code transformations, the hardest problem in automating loop flattening is to determine its *safety*. A sufficient condition is that the loop into which we lift an inner loop body can be parallelized, which might be hard to detect, especially if indirect addressing occurs. However, this is already a necessary condition for parallelizing loops in general, and therewith a standard problem for parallelizing compilers [HKT92a]. The same technology developed there can be applied here.

When safety is ensured, either by user information (like a FORALL loop header) or by "heroic dependence analysis," we expect that the systematic loop flattening transformation, as described in Section C.3, can be implemented efficiently into compilers like the FORTRAN D compiler in the ParaScope programming environment [KMT91]. This implementation should not be part of this dissertation; what it will contain, however, is a performance study on which improvements to be gained when applying loop flattening manually and an evaluation of its applicability and automatability.