

# Local and Global Data Distribution in the Filaments Package \*

David K. Lowenthal  
Department of Computer Science  
The University of Georgia  
Athens, GA 30602  
dkl@cs.uga.edu  
<http://www.cs.uga.edu/~dkl>

**Abstract** *It is generally agreed that using threads and shared memory provides a desirable parallel programming model. However, to achieve scalability it is often necessary to execute these programs on a distributed-memory multicomputer. The Filaments package provides an interface of threads and shared memory with an implementation on a distributed memory machine through a software distributed shared memory (DSM). This focus of this paper is on the problem of finding effective data (thread) distributions both within and between phases, taking data redistribution into account, in scientific applications composed of multiple phases. Our approach, which is implemented within the Adapt run-time data distribution system, takes measurements on the first iteration of the outermost loop in the application, and uses them to find the global distribution (over a reasonable set of distributions) that leads to the best completion time. Initial results are encouraging; for a flame code simulation where behavior is largely dependent on input data, Adapt finds an effective global data distribution with reasonable overhead.*

*Keywords:* threads, data distribution, distributed shared memory, load balancing, communication

## 1 Introduction

One way to write a program for a parallel computer is to use a thread for each independent unit of work and shared variables for (implicit)

communication. Although this model provides a simple way to program, it requires an efficient implementation to be a viable option on a distributed-memory multicomputer. The Filaments package [1, 2, 3] provides an efficient implementation of threads and shared memory on distributed-memory machines. Efficiency in Filaments is obtained through stackless threads and a multithreaded software distributed shared memory (DSM).

Although a DSM system provides implicit communication between nodes, it is purely focused on *when* to communicate; *how much* program data must be communicated and *how well* the computational load is balanced are largely orthogonal problems. A multithreaded DSM system such as Filaments does not in any way prevent threads from being distributed to nodes such that one or both of the following occurs:

- threads reside on one node but access data on a *different* node, or
- threads on one node perform more work than threads on other nodes.

In fact, in a DSM system, the distribution of threads corresponds to the general data distribution problem on multicomputers, which is to distribute data such that communication is minimized and the computational load is balanced. This is because threads incur page faults, which result in page-based communication, and also perform all computation, so imbalances in thread workloads lead to delays

---

\*This research was supported by NSF CAREER Grant CCR-9733063. Paper appeared in PDPTA '98.

at barrier synchronization points. For the rest of this paper, we will use the term data distribution, with the understanding that in a thread-based DSM system such as Filaments, the threads are actually distributed.

Data distribution is a difficult problem on which much prior research has been performed (see, for example, language approaches such as [4, 5, 6], compiler approaches such as [7, 8, 9], run-time approaches such as [10, 11, 12, 13], and integrated compiler/run-time approaches [14]). The run-time approaches focus on the distribution of data in a single program phase. This contribution of this paper is the development of a run-time approach to the global data distribution problem, which means that we find at run time a distribution of each array in each program phase, with possible redistribution. We perform this in the context of the Filaments package through the Adapt run-time data distribution system [10]. Although our work is done in the setting of the Filaments package, it is directly applicable to any DSM system and is largely applicable to the general data distribution problem.

The rest of this paper is organized as follows. Section 2 gives a brief description of the Filaments package as it relates to this paper. Section 3 describes our general framework. Section 4 discusses how we find data distributions in Filaments for a specific phase of an application, and Section 5 explains how we find global data distributions in Filaments. Finally, Section 6 gives performance results and Section 7 summarizes the paper and discusses avenues for future work.

## 2 The Filaments Package

The Filaments package allows efficient parallel programs to be written using a thread (filament) for each independent unit of work and a global shared memory. A *filament* is a very lightweight thread that references a shared address space. Although generally thought to be inefficient, such *fine-grain* programs have several advantages. First, they are architecture

independent in the sense that parallelism is expressed in terms of the application and problem size, not in terms of the number of processors that might actually be used to execute the program. This also makes fine-grain programs easier and more flexible to write, because it is not necessary to cluster independent units of work into a fixed set of larger tasks. Third, the inner-loop parallelism extracted by parallelizing compilers such as SUIF [15] or expressed in parallel variants of languages such as Fortran are naturally fine-grain. Finally, when there are many more processes than processors, it is often easier to balance the total amount of work done by each processor.

Filaments supports a global address space through a software distributed shared memory (DSM). The Filaments DSM is similar to other software DSM systems (see, for example, [16]); it is implemented on top of UNIX using system calls and sockets. It is unique in its support for multithreading; several filaments can execute on each node, allowing the latency of page faults to be mitigated.

This Section gives a brief overview, using Jacobi iteration pseudocode (shown in Figure 1) as a single example, of the parts of Filaments that are relevant to this paper. More information on Filaments can be found in [1, 2, 3].

The `jacobi()` function simply updates a single point of an array based on a nearest-neighbor calculation. This application uses iterative filaments, which execute repeatedly, with barrier synchronization and termination detection occurring after each execution of all filaments. The programmer or compiler must specify a function that is executed on each node after all filaments have completed. In this example `terminate()` returns `F_DONE` if `numIters` iterations have been performed, and `F_CONTINUE` otherwise.

The main function places arrays in DSM memory and then initializes them. It is also responsible for creating the filaments. The code runs on each node, so functions `getStart()` and `getEnd()`, which return a unique range for  $i$  on each node, determine which filaments execute on which nodes. The distribution of

```

double **old, **new; int k = 0, numIters = ...;
void jacobi(int i, int j) {
    double temp;
    new[i][j] = (old[i-1][j] + old[i+1][j] + old[i][j-1] + old[i][j+1]) * 0.25;
}
int terminate() {
    if (k > numIters)
        return F_DONE;
    return F_CONTINUE;
}
void main() {
    int i, j, n = atoi(argv[1]);
    /* place arrays old and new into DSM memory and initialize them */
    for (i = getStart(); i <= getEnd(); i++) { /* compute start, end on each node */
        for (j = 1; j < n-1; j++)
            filCreateIterFilament(jacobi, i, j); /* each filament calls jacobi w/ parms. i and j */
    }
    filStartFilaments();
}

```

Figure 1: Jacobi iteration outline

filaments to nodes determines the distribution of data via DSM page faults. In this example,  $(n - 2)^2$  filaments are created; each can be identified uniquely by its  $(i,j)$  coordinates. All filaments in row  $i$  access rows  $i - 1$ ,  $i$ , and  $i + 1$  of the *old* array and row  $i$  of the *new* array on each iteration. Assuming all pages initially reside on a single node (numbered 0), filaments on all other nodes will cause initial page faults to acquire the needed data. On all subsequent iterations, only filaments that update boundary values of the array (the top and bottom rows on each node) will cause page faults.

### 3 Framework

This section describes our general framework for the data distribution problem. First, we assume that an application uses regular meshes that result in regularly accessed arrays, as opposed to irregular meshes that result in either indirectly accessed arrays or linked data structures. Second, we consider data distributions where the first dimension is distributed (in a DSM system, distributing more than one dimension requires significant array restructur-

ing). We also assume an application is composed of *phases*. Informally, a phase is application code terminated by a barrier synchronization point. Precisely, a phase is the outermost loop of a loop nest that contains no barrier synchronization points. For example, consider the outline of a code to perform a simulation of hydrocarbon flames, shown in Figure 2 and adapted from [17]. There are two phases, both contained within the outermost loop; the first consists of the first three lines, and the second consists of the last two lines. Both loops must be separate phases because barrier synchronization is necessary between the loops to ensure consistency of arrays  $x$  and  $z$ . Intuitively, phase boundary points are potential places to redistribute data, as array use generally does not change within a phase. Our framework allows redistribution only between phases.

We are interested in finding a *global data distribution*, which is a distribution of each array in each phase, with possible redistribution between phases. The ideal global distribution minimizes the overall completion time of an application. Note that because all nodes coop-

```

for time := 1 to timesteps {
  for i := 1 to N
    x[i] := x[i] + F(y[i-1],y[i],y[i+1],z[i])
    y[i-1] := x[i-1]
  for i := 1 to N
    z[i] := AdaptiveSolver(x[i])
}

```

Figure 2: Flame code outline. Note that there are two phases and the *AdaptiveSolver* function has a workload that is dependent on the value of its parameter.

erate in order to complete an application, the completion time of the slowest node determines the completion time of the application.

Three factors affect the completion time of a node: computation time, communication overhead, and synchronization delay. Computation time is essentially the time spent executing application code. Communication overhead is both time spent executing low-level code that copies messages (page requests and replies) to and from the network and time spent waiting for pages from other nodes (if necessary). Synchronization delay is time spent waiting for other nodes to complete their computation. This section describes our models of computation and communication and how a distribution affects computation time, communication overhead, and delay. Then we discuss finding both intra- and inter-phase data distributions.

### 3.1 Computational Model

Our computational model is Single Program Multiple Data (SPMD) [4], in which each node executes the same code but references a different subset of the data elements. In our framework this means that all filaments on a node execute the same code. We also require iterative computations, because we will be making distribution decisions after monitoring one iteration of an application.

Although any node can potentially update any data element, we assume the *owner-computes* rule [4]. This rule states that each data element has an “owner” node and that is

the only node that will update the element; however, other nodes may reference the element.

The data distribution affects both computation time and communication overhead. Because of the owner-computes rule, the number of filaments (and hence data elements) each node owns, along with the amount of time to update each element, determines the time it spends computing. Communication occurs when the currently executing filament on a node incurs a page fault while updating a local data element; therefore, the distribution of filaments (and hence data) also determines the communication overhead on a node.

Both the computation time and communication overhead on a node can lead to synchronization delay. If the total computation and communication between the nodes is unbalanced, they will finish at different times, causing the early finishing nodes to block at a barrier waiting for the others to finish. The key for a good data distribution is balancing the computation between the nodes—to minimize synchronization delay—while also minimizing the number of page faults—to minimize communication overhead.

### 3.2 Distribution Strategies

Within a phase, the elements of a data structure can be distributed to the nodes in numerous ways. However, the goals of simultaneously balancing computational load and minimizing communication often conflict, as there

is an interaction between the two. For problems with regular mesh structures, two primary types of distributions are *variable block* and *striped*. A variable block distribution distributes a contiguous set of data elements to each node. The sizes of the blocks are chosen so that an equal amount of work is done on each node. For applications with regular, nearest-neighbor access patterns, such as Jacobi iteration, communication occurs only on edges with a block-based scheme. (Note that if the blocks are equal, this distribution is referred to as BLOCK.) The other type of distribution, where data is *striped* across the nodes, is called CYCLIC; it is good for applications with decreasing amounts of work and a distribution-independent communication pattern, such as LU decomposition.

As difficult as it can be to find the right data distribution for a single phase, finding a good global data distribution is even more difficult. Some applications that have multiple phases have a single distribution that works well for all phases. However, it can be hard to find a good global distribution even for seemingly simple programs. Consider again the flame code. There are two phases, with nearest-neighbor communication in the first and no communication in the second. Furthermore, the workload is uniform in the first phase, so for best performance, the arrays in the first phase,  $x$ ,  $y$ , and  $z$ , should use a BLOCK distribution. However, the best distribution for the second phase is unknown at compile time because the work involved in performing the *AdaptiveSolver* function is dependent on the value of its parameter ( $x[i]$ ). While using a BLOCK distribution avoids any redistribution overhead, it could cause a severe load imbalance in the second phase. A variable block distribution can balance the load while avoiding false sharing that would arise from redistributing only single (expensive) points. However, because the workload is unknown, the block-size cannot be determined at compile time.

Fortunately, the costs associated with the *AdaptiveSolver* vary gradually as the computation progresses, so run-time information about

the second phase of iteration  $i$  can be used to determine a good distribution for the second phase of iteration  $i + 1$ . However, it is still not straightforward to determine a good global data distribution. To decide on a distribution for the *entire* program, one must decide between choosing the most effective distribution for one of the phases, a less effective distribution for both phases, or the most effective distribution in each phase with a redistribution. The first two avoid any cost due to redistribution of data. However, this means that at least one of the phases will execute with a suboptimal distribution. The latter choice ensures that each phase itself has the most effective distribution at the cost of a redistribution.

## 4 Local Data Distribution

This section describes how we choose an effective data distribution within a phase in Adapt. The Adapt prototype is implemented on top of Filaments. The Adapt system dynamically selects either a variable block or striped distribution for a single application phase. Each phase starts by using some initial data distribution by the programmer or compiler (the current default is BLOCK) and then employs two steps—instrumentation and selection—to determine whether this distribution is a good one or whether it should be changed.

### 4.1 Instrumentation

Adapt gathers information about the communication and computation in each loop body. Adapt monitors communication using DSM page faults and the DSM page table. It determines the communication pattern by inspecting the pattern of page faults on arrays in the page table. Currently, Adapt recognizes two patterns: *nearest-neighbor* and *broadcast*. In the nearest-neighbor pattern, node  $i$  needs to communicate values with nodes  $i + 1$  and  $i - 1$ . This pattern occurs on an array when (1) each node has a distinct subset of exclusive-access pages of the array and (2) neighboring nodes

have read access to consecutive sets of pages of the array, with each node owning one set.

A broadcast pattern occurs when one node writes a value, there is a barrier, and then all nodes read the value. Adapt detects a broadcast pattern on an array if there are a pair of loops that exhibit the following characteristics: (1) in the first loop one node writes to a subset of pages of the array, (2) in the second loop each node has a distinct subset of exclusive-access pages of the array, and (3) in the second loop all nodes read the subset of pages that were written in the first loop.

Adapt gathers information about computation time by instrumenting the application code to obtain the time each node spends updating data elements it owns. These times are combined at the next barrier synchronization point to obtain the total computation time.

## 4.2 Selection

After gathering communication and computation information for one iteration of an application, Adapt uses it to choose a good data distribution. In particular, given the total computation time  $T$  and the number of nodes  $P$ , the ratio  $T/P$  represents the amount of computation each node should perform for a perfectly balanced load. Adapt examines different ways that rows could be mapped to nodes to achieve this ideal load. This depends on the communication pattern detected during the monitoring phase. When the communication pattern is nearest-neighbor, Adapt maps a consecutive number of rows to each nodes so that the estimated total time on the node is as close as possible to  $T/P$ . When the communication pattern is broadcast, Adapt assigns rows to nodes in a cyclic manner, because the communication is independent of the number of edges.

## 5 Global Data Distribution

The last section discussed how to find a good intra-phase, or local, distribution. This section focuses on how Adapt finds an inter-phase, or global, distribution. Assuming a program has

been divided into its component phases by a programmer or compiler, the goal of Adapt is to find an effective global data distribution. This is done by executing an outermost loop one time and using the per-phase run-time information to choose a global distribution. The data will be redistributed accordingly before execution of the next iteration.

The general idea of how we handle global data distribution follows. At the end of each phase, the two steps described in the previous section are performed. Now, consider a point at which the last phase within an outer loop has completed, and the outer loop is not contained in some phase (this will often be the outermost loop in the program). At this point, all phases within the loop have been executed once. Adapt has already found an effective local distribution for each phase, along with its estimated completion time. Adapt also retains the timing measurements so that it can estimate completion times for less effective variable block and striped distributions. Suppose there are  $k$  phases that were executed on the first iteration of an application. Denote the most effective distribution for phase  $i$  as  $D_i$ , along with its estimated completion time  $T_i$ . (Note that each  $D_i$  represents the distribution for *all* arrays in that phase. This is because we are actually distributing filaments, while *all* data moves in response to page faults—the distribution of a single filament results in the distribution of every array it accesses.) The estimated completion time for an iteration of the outermost loop containing  $k$  phases (using distribution  $D_i$  for phase  $i$ ) is  $\sum_{i=1}^{k-1}(T_i + R_{i,i+1}) + T_k + R_{k,1}$ , where  $R_{i,i+1}$  is the time to redistribute data from phase  $i$  to phase  $i+1$ . Due to redistribution overhead, using the most effective local distribution in each phase might not be effective globally.

It is clear that we must take the potentially high redistribution costs into account. We assume that redistribution costs are computed for a given architecture and are known. Adapt considers each distribution that was most effective in some phase to be a candidate. To find an effective global distribution, Adapt will

create a graph that we call the run-time data distribution graph, or RDDG (which is similar to the decomposition graph described by Kennedy and Kremer [18]). The RDDG has  $k$  rows of nodes, one for each phase. Each row will consist of at most  $k$  nodes, one for each candidate, so there will be  $O(k^2)$  nodes in the graph. A node can be identified by a pair  $(i, j)$ , where  $i$  is the phase (row) and  $j$  is a candidate. The graph has an edge between each pair of nodes  $(i, j_1)$  and  $(i + 1, j_2)$ , where  $i$  is the row and  $j_1$  and  $j_2$  specify candidates. The edge is weighted by the time to execute phase  $i$  using distribution  $j_1$  plus the time to redistribute the data from distribution  $j_1$  to  $j_2$ . We also add a copy of the first row at the bottom of the graph to capture potential redistribution back to the first phase. At this point we find the shortest path through the RDDG from *each* of the  $k$  nodes in the first row (the first phase) to its corresponding node in the last row (which is a copy of the first phase). We must find  $k$  shortest paths and the graph has  $O(k^2)$  nodes and  $O(k^3)$  edges; hence, the overall time to find the best distribution will be  $O(k^4)$ .

After the best distribution for each array in each phase is found, the distribution for each phase is modified. Adapt changes the data distribution by moving filaments to effect a reparameterization of the code. When filaments on a node accesses data they do not own, page faults result; the underlying DSM then moves the data. After a distribution has been changed, Adapt continues to monitor the application in a coarse-grain manner. If it detects a large variance in computation times or an increase in communication times, the monitoring is re-enabled and the global data distribution is again computed.

## 6 Performance

This section reports the performance of Adapt on four programs: Jacobi iteration, LU decomposition, particle simulation, and flame simulation. The first three are applications that

each have a single data distribution that is effective globally. Jacobi iteration and LU decomposition are examples of applications for which it is possible to determine a good data distribution statically; the best distribution for Jacobi is BLOCK in each phase, and the best for LU is CYCLIC in each. Particle simulation, on the other hand, requires run-time support both to determine a good distribution and possibly to change the distribution during the computation, because the initial particle positions and how they cluster are unknown. Its best distribution will be blocks of varying size in each phase, depending on the particle positions. Still, this variable block distribution will be the *same* in each of its two phases.

On the other hand, the flame code is composed of two phases, where each each has a *different* most effective distribution. Hence, the input data determines whether the best global data distribution is a different distribution for each phase with redistribution in between or one of the (same) suboptimal distributions in each phase.

For each application we developed a program using Adapt. For an accurate comparison, we also developed a Filaments program, without the Adapt subsystem, that chose a (predetermined) static distribution. (Sequential programs were virtually identical to the one-node Filament/Adapt programs.) Below, we present the results of runs on 1, 2, 4, and 8 Sparc-1 nodes connected by an Ethernet. Our implementation of global data distribution within Adapt is work in progress; to date we have executed the flame code on 2 and 4 Pentium Pro nodes on connected by a Fast Ethernet. All programs used gcc with the `-O` flag.

### 6.1 Single-Distribution Applications

The execution times for the three single-distribution applications are shown in Figure 3. All Adapt programs initially use BLOCK unless otherwise specified. For the Adapt version of Jacobi iteration, after nearest-neighbor communication is recognized, the block resizing reproduces the BLOCK distribution. This is be-

Number of Nodes	1	2	4	8
Jacobi: Adapt Time (sec)	189	104	55.2	32.0
Jacobi: Filaments Time, <b>BLOCK</b> (sec)	188	104	54.6	30.4
LU: Adapt Time ( <b>BLOCK</b> ) (sec)	547	322	210	185
LU: Adapt Time ( <b>CYCLIC</b> ) (sec)	547	305	190	165
LU: Filaments Time, <b>CYCLIC</b> (sec)	544	303	189	164
Particle Sim.: Adapt Time (sec)	69.4	40.1	29.8	23.5
Particle Sim: Filaments Time, <b>Best BLOCKCYCLIC</b> (sec)	69.1	46.5	34.2	25.3

Figure 3: Performance of Jacobi iteration, LU Decomposition, and Particle Simulation. Jacobi iteration uses a  $512 \times 512$  grid and  $\epsilon = 10^{-3}$ . LU decomposition uses an  $800 \times 800$  grid, and the particle simulation uses a grid of  $64 \times 64$  with 150 particles.

cause the work at each point is uniform.

There are two Adapt LU programs; one uses **BLOCK** as its initial distribution, and the other uses **CYCLIC**. For the former, after a broadcast communication pattern is recognized, Adapt changes to a **CYCLIC** distribution; this causes page faults necessary to change the distribution and is significant, causing up to a 12% overhead. The second program, Adapt **CYCLIC**, initially uses a cyclic distribution; because there is no redistribution overhead, the performance is similar to the Filaments program.

In our particle simulation, the particles tended to move to the upper region of the grid. The Adapt version performs the best in this case, because when more particles cluster near the top, Adapt remaps the space array to balance the number of particles. The Filaments program with the best possible **BLOCKCYCLIC** distribution is shown as comparison; however, larger blocks cause more load imbalance, and smaller blocks cause more communication. Note that a (static) variable block distribution can be used, but unless one has prior knowledge of particle behavior, it is no better than **BLOCK** in general.

## 6.2 Flame Code

Due to time constraints, we were not able to fully implement the flame code described in Section 3; this is still in progress. To test our prototype, however, we wrote a code that mimics the structure of the flame code. Our program contains two phases; the first performs

a simple nearest neighbor calculation, and the second is dependent only on local points (just as shown in the flame code in Figure 2). Both phases contain parameterizable delay loops so that we could experiment with differing phase execution times.

Figure 4 shows the performance of our modified flame code on 2 and 4 nodes. We ran tests with three different delay loops; all were such that in the second (adaptive) phase, the work was concentrated in the top quarter of the grid. The first test weighted the second phase heavily, while the second and third tests weighted both phases evenly. The second test contained a small amount of work in each phase, while the third contained a large amount in each. The best distributions for the tests were variable block in both phases (first test), block in both phases (second test), and block/variable block with redistribution (third test). The runtime analysis of Adapt successfully found these distributions with small overhead compared to the Filaments programs (where we used prior knowledge). A good global distribution is vital to good performance; for example, when using a poor distribution, the first flame test actually *slowed down* when moving from two to four nodes.

## 7 Summary and Future Work

We have described a threads package called Filaments that uses fine-grain threads and a distributed shared memory (DSM). An impor-

Number of Nodes	1	2	4
Flame (1): Adapt Time (sec)	177	115	75.5
Flame (1): BB/BV/VV (sec)	177	131/147/113	100/187/72.6
Flame (2): Adapt Time (sec)	198	121	77.8
Flame (2): BB/BV/VV (sec)	198	120/150/145	77.5/127/100
Flame (3): Adapt Time (sec)	144	89.0	61.4
Flame (3): BB/BV/VV (sec)	144	92.0/88.8/118	64.1/59.4/103

Figure 4: Performance of 3 versions of our flame code, size  $1024 \times 1024$ . BB indicates that each phase used **BLOCK**, BV indicates that the first phase used **BLOCK** and the second used variable-sized blocks, and VV indicates both phases used (the same) variable-size blocks.

tant problem in a DSM is how to distribute the threads (filaments) to the nodes. Such a decision must take into account both workload and communication. We showed how Adapt makes intra- and inter-phase (global) distribution decisions. Initial results are encouraging, as Adapt finds, with no prior knowledge, an effective global distribution for the flame code. Furthermore, it does so with reasonable overhead.

Our work on the global data distribution problem is just beginning. Finding good global data distributions is a difficult problem. We assumed that the global distribution consists of distributions that were effective in at least one phase. We are working on allowing global distributions where each phase is suboptimal; the challenge is to avoid an explosion of nodes in the RDDG. We are also working on handling more complex program patterns, such as those with conditionally executed and nested phases. Finally, we are integrating a compiler with our run-time system.

## References

- [1] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–212, November 1994.
- [2] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37:41–54, November 1996.
- [3] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Efficient support for fine-grain parallelism on shared-memory machines. *Concurrency: Practice and Experience*, 10(3):157–173, March 1998.
- [4] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communication of the ACM*, 35(8):66–80, August 1992.
- [5] H.P. Zima, H.J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(6):1–18, January 1988.
- [6] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.
- [7] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 424–432, October 1990.
- [8] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 213–223, April 1991.
- [9] M. Gupta and P. Banerjee. PARADIGM: A compiler for automated data distribution on multicomputers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 357–367, July 1993.
- [10] David K. Lowenthal and Gregory R. Andrews. An adaptive approach to data placement. In *Proceedings of the 10th International Symposium on Parallel Processing*, pages 349–353, April 1996.
- [11] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, August 1995.
- [12] Yuan-Shin Hwang, Bongki Moon, Shamik D. Sharma, Ravi Ponnusamy, Raja Das, and Joel H. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed-memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.
- [13] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.
- [14] Sotiris Ioannidis and Sandhya Dwarkadas. Compiler and runtime support for adaptive load balancing in software distributed shared memory systems. In *Proceedings of the Fourth Workshop on Languages, Compiler, and Run-Time Systems for Scalable Computers (to appear)*, May 1998.
- [15] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau-Wen Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.
- [16] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.
- [17] HPP-2 scope of activities and motivating applications. November 1994.
- [18] Ken Kennedy and Ulrich Kremer. Automatic data layout for High Performance Fortran. Technical Report CRPC-TR94498-S, Rice University, December 1994.