An Integrated Compiler/Run-Time System for Global Data Distribution in Distributed Shared Memory Systems*

Gregory M.S. Howard†
David K. Lowenthal‡
Department of Computer Science
The University of Georgia
Athens, GA 30602

Abstract

A software distributed shared memory (DSM) provides the illusion of shared memory on a distributed-memory machine; communication occurs implicitly via page faults. For efficient execution of DSM programs, the threads and their implicitly associated data must be distributed to the nodes to balance the computational workload and minimize communication due to page faults. The focus of this paper is on finding effective data distributions in DSM systems both within and across all computational phases. Our model takes into account data redistribution between phases. We have designed and implemented an integrated compiler/run-time system called SUIF-Adapt. The compiler, which is an extended version of SUIF, divides the program into phases, analyzes each, and communicates important information to the run-time system. We use an extended version of Adapt, a run-time data distribution system, to take measurements on an iteration of a loop consisting of one or more phases. It then finds the global data distribution *for the loop (over a reasonable set of distributions) that leads to the best completion time. Performance results indicate that programs that use SUIF-Adapt can outperform programs with predetermined data distributions when phase behavior is dependent on run-time values of input data; in such cases, statically determining an effective data distribution requires (generally unavailable) prior knowledge of run-time behavior of an application.*

*This research was supported by NSF `CAREER` Grant CCR-9733063.

†Current address: Silicon Graphics Inc., 655F Lone Oak Drive, Eagan, MN 55121. Email: ghoward@sgi.com

‡Corresponding Author. Email: dkl@cs.uga.edu

1 Introduction

Distributed shared memory (DSM) systems eliminate explicit internode communication by providing the abstraction of shared variables. Although a DSM transparently manages communication, good performance still depends on balancing the computational load and minimizing communication. In a page-based DSM system, threads perform the computation and cause communication (page faults) when they reference nonlocal data. Hence, in a DSM system, the distribution of threads corresponds to the general *data distribution problem*, which is to distribute data such that communication is minimized and the computational load is balanced. Unfortunately, finding effective data distributions statically is not possible for applications whose data access patterns or workload characteristics exhibit unpredictable or dynamic behavior.

This paper presents an integrated compiler and run-time approach to the *global data distribution problem* in DSM systems, which is to find a series of data distributions for program *phases* that minimizes overall completion time, including any time required to redistribute data. We define a phase as a section of application code between two barrier synchronization points. We have extended the SUIF parallelizing compiler [HAA+96] with passes that (1) locate groups of phases and possible redistribution points, (2) choose an initial data distribution for each phase, and (3) generate code with calls into the run-time system at appropriate points. Our run-time system uses a multiple-phase model to calculate efficient global data distributions. It extends the single-phase model of Adapt [LA96], which finds effective local distributions in DSM systems.

Performance results indicate that programs that use SUIF-Adapt perform comparably with programs that use predetermined ("hand-coded") data distributions.

This includes static and dynamic applications. In all cases, the execution time of the program using SUIF-Adapt was always within 7% of the *best* hand-coded version. Further, when a phase's workload characteristics change dynamically, SUIF-Adapt can perform considerably better than the hand-coded program. Also, it is important to note that some of the hand-coded programs make use of *prior knowledge* of run-time behavior of an application; in general, such information is unavailable statically.

The remainder of this paper is organized as follows. Section 2 describes our framework. Section 3 provides details on the implementation of our SUIF-Adapt system. Section 4 presents the results of performance tests of four different programs, while Section 5 discusses areas of current research. Section 6 discusses previous work in this area, and Section 7 gives some concluding remarks.

2 Framework

In this section we describe the computational model upon which our system is based and the data distribution strategies that our system uses.

2.1 Computational Model

Our computational model is Single Program Multiple Data (SPMD) [HKT92], in which each node executes the same code but references a different subset of the data elements. The applications we currently address use regularly-accessed arrays and are divisible into one or more *phases*, which are sections of application code between two barrier synchronization points. We also only consider data distributions where the first dimension is distributed. This is because in our DSM system, distributing more than one dimension requires significant array restructuring. Furthermore, we assume the existence of a loop that directly encloses one or more phases; we call this a *phase cycle*. A sample application (Jacobi iteration) is presented in Figure 1.

Although any node can potentially update any data element, we assume the *owner-computes* rule [HKT92]. Under this rule, each node is exclusively responsible for updating a distinct subset of data elements; the number of threads (and hence data elements) each node owns therefore determines the time a node spends computing. On the other hand, the time a node spends on internode communication is determined by the number of DSM page faults incurred. The key for a good data distribution is avoiding large variances in completion times between nodes while also minimizing the number of page faults. This should be done such that the total execution time of the slowest node is minimized, because each phase is terminated with a barrier synchronization point—no node can continue until all have arrived. In other words, the goal is to minimize the maximum execution time of any node.

2.2 Distribution Strategies

For problems with regular arrays, two common types of distributions are *variable block* and *striped*. A variable block distribution distributes a contiguous set of data elements to each node, where block sizes vary in order to balance the workload. For applications with regular, nearest-neighbor access patterns, such as Jacobi iteration, a block-based scheme requires communication only on block boundaries. (Note that if the block sizes are equal, this distribution is referred to as `BLOCK` [FHK+90].) The other type of distribution, where data is *striped* across the nodes, is called `CYCLIC` [FHK+90]; it is good for applications with increasing or decreasing amounts of work per iteration and a distribution-independent communication pattern, such as LU decomposition.

These distributions apply to a single phase; we seek a series of such *local* distributions that will provide good performance for the entire *phase cycle*. Consider an outline of a flame simulation shown in Figure 2. There are two phases, with nearest-neighbor communication in the first and no communication in the second. The workload is uniform in the first phase, so a `BLOCK` distribution for all arrays is desired; our compiler can infer this. However, the best distribution for the second phase must be determined dynamically because the workload of *AdaptiveSolver* depends on the value of $x[i]$; it will be some variable-block distribution. For the phase cycle, we must decide between using one of the best local distributions for both phases, a less effective distribution for both phases, or the most effective distribution in each phase with a redistribution. The first two avoid data redistribution at the cost of executing one or more phases in a suboptimal distribution. The latter choice ensures that each phase executes with its own most effective distribution at the cost of a redistribution.

3 Implementation

Our integrated SUIF-Adapt system is composed of two subsystems: a modified version of the SUIF compiler [HAA+96] and an extended multiple-phase version of Adapt [LA96], a run-time data distribution system. The Adapt system runs on top of Filaments [LFA96]. Filaments provides a simple model for writing or generating efficient, portable parallel programs.

```
for step := 1 to numsteps {
  for i := 1 to N
    for j := 1 to N
      y[i] = 0.25 * (x[i-1][j] + x[i+1][j] +
                     x[i][j-1] + x[i][j+1]);
  for i := 1 to N
    for j := 1 to N
      x[i] := y[i];
}
```

Figure 1: Jacobi iteration outline. Each (entire) `for i` loop is a phase (barrier synchronization is required after each loop, but not within the loop bodies); the `for step` loop is a phase cycle.

```
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+1])
  for i := 1 to N
    z[i] := AdaptiveSolver(x[i])
}
```

Figure 2: Flame simulation outline. The *AdaptiveSolver* function has a workload that is dependent on the value of its parameter. Again, each `for i` loop is a phase.

To achieve this goal, it provides efficient fine-grain parallelism and a multithreaded DSM. However, in this work we do not overlap communication and computation. The Filaments DSM provides causal consistency, which is slightly weaker than sequential consistency. However, it does not allow a write-shared protocol; we use write-invalidate in this work, which makes the avoidance of false sharing important. Adapt requires the following features from a DSM: an accessible page table, facilities to measure the time of each row, and access to the barrier synchronization mechanism. Filaments provides all of these items.

3.1 Modifications to SUIF

We have modified SUIF to find phase cycles (it already finds phases), perform lightweight analysis of the workload and communication characteristics of each phase, and generate Adapt and DSM-compatible code. The first SUIF modification, finding phase cycles, is straightforward and done by searching for sequential loops that contain phases.

Our lightweight phase analysis attempts to detect common communication patterns, classifying each phase as *nearest neighbor*, *broadcast*, *no communication*, or *unknown* based on shared array accesses and loop indices. Our compiler does not do comprehensive dependence analysis; rather, it looks for commonly occurring characteristics of the above patterns. If it misses an opportunity to detect a pattern, Adapt will detect it at run time.

Nearest-neighbor is detected in a phase when (1) the phase contains a read to an array, (2) a different phase contains a write to the array, (3) the index expressions differ by a constant, and (4) the loop bounds in each phase are the same. Two phases are marked as broadcast when the first is a sequential phase that contains a write to a shared array, and the second is a parallel phase where all nodes read the data that was written in the first phase. Finally, phases that operate only on local data are marked as no communication.

This pass also characterizes workload uniformity on the basis of nested loop bound interdependencies. A loop whose lower or upper bound is an affine function of the index variable of the phase cycle causes the inner loop to be marked as having an increasing (or decreasing) workload [GB92]. If the loop bounds are uniform, the interior of the loop is checked for conditional statements. If there are none, the loop itself is uniform.

Finally, our modified compiler inserts calls to the Adapt and Filaments run-time systems. These include

run-time analysis routines (as described below) at the end of each phase and phase cycle, calls that choose the initial distribution for each phase, and a barrier between each phase. The initial distribution is chosen in a similar way as done in [GB92] and is based on the phase analysis described above.

3.2 Adapt

The Adapt run-time system [LA96, Low98] monitors program execution and, if necessary, chooses and implements new local (per-phase) and global data distributions at run-time. This section first describes the monitoring of each phase to choose an effective per-phase distribution and then describes the method used to choose an effective global data distribution.

3.2.1 Instrumentation and Local Selection

Initially each phase uses the compiler-chosen distribution (`BLOCK` or `CYCLIC`). Adapt generally measures the computation time of each row of the array being updated on the second phase cycle iteration in order to ensure that execution time measurements are not skewed by page faults effecting the initial distribution. If the compiler can guarantee that the data is already distributed on entrance to a phase cycle, monitoring takes place on the first iteration.

As described below, Adapt in most cases selects a distribution for each phase based on the results of run-time monitoring of execution times and the communication pattern inferred by the compiler; however, for phases whose communication pattern is *unknown*, Adapt will also attempt to infer the communication pattern using the pattern of faults in the page table (for details, see [LA96]).

After gathering computation and (possibly) communication information for each phase, Adapt selects an effective local distribution for that phase. When the pattern is nearest-neighbor, Adapt chooses a distribution of consecutive rows to each node (variable blocks) so that the estimated total time on the node is as close as possible to T/P, where T is total computation time and P is the number of nodes. When the communication pattern is broadcast, Adapt chooses a cyclic distribution so that the load will be balanced, since the amount of communication in such phases is constant (over all row-based distributions).

3.2.2 Global Data Distribution

When the first iteration of the phase cycle is complete, the information gathered for each phase along with the cost of page faults in the Filaments DSM is used to find the most effective *global* distribution. Adapt computes projected execution times for each phase in each (row-based) best local distribution (since we have the time for each row, this can be done by summing over all rows that would be assigned to a node). It uses these times as well as the communication costs associated with redistribution to arrive at the most effective global distribution. This is accomplished by representing the phase cycle as a graph that we call the run-time data distribution graph, or RDDG (very similar to the decomposition graph described by Kennedy and Kremer [KK98]). An example RDDG for the flame simulation is shown in Figure 3. In the case that each phase represents a distribution for every array, the problem can be solved by finding the shortest path [KK98]. Otherwise, the problem is NP-complete; we use a simple greedy heuristic, where the shortest path between two phases is computed based on the arrays accessed in the destination phase. All arrays not accessed in the phase are determined to be in the same distribution as they were in the source phase. After the best distribution in each phase is found, each node modifies its loop bounds in each phase to effect the new global distribution (via DSM page faults).

Further Details

Our SUIF-Adapt system contains both static and dynamic analysis. One challenge is for our system to be as efficient as a static-only system on very regular applications. However, the dynamic nature of our system presents opportunities to make several improvements over a static-only, compiler-based system. Several of these improvements are briefly discussed below and shown in Figure 4; space limitations prevent further discussion.

First, we support applications that require periodic rebalancing, such as particle simulation. Once a new global distribution is chosen, Adapt continues coarse-grain monitoring, where only the time per node is measured. If the load becomes unbalanced, a new global distribution is found by reapplying the steps outlined in Section 3.2. This is necessary to support applications like particle simulation. Second, situations may arise where the cost of redistributing data in a "better" global distribution might actually *outweigh* the cost of running the program to completion with no redistribution. SUIF-Adapt avoids this situation when the number of phase cycle iterations is static by maintaining the number of remaining iterations and augmenting the RDDG with a new "start row" representing

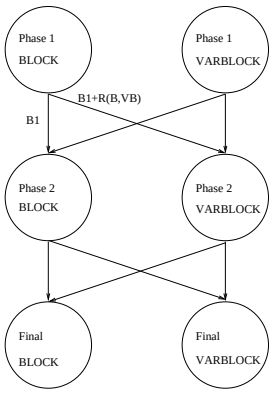

Figure 3: RDDG for flame simulation. An edge between two nodes represents the cost of executing the phase represented by the source node in the given distribution plus the cost, if any, of redistributing to the phase represented by the destination node. For example, $B1$ denotes the time to perform phase 1 using `BLOCK`. $R(B, VB)$ denotes the time to redistribute from `BLOCK` to `VARBLOCK`. For clarity, most of the edge weights are omitted. The shortest path through this graph is the best global distribution.

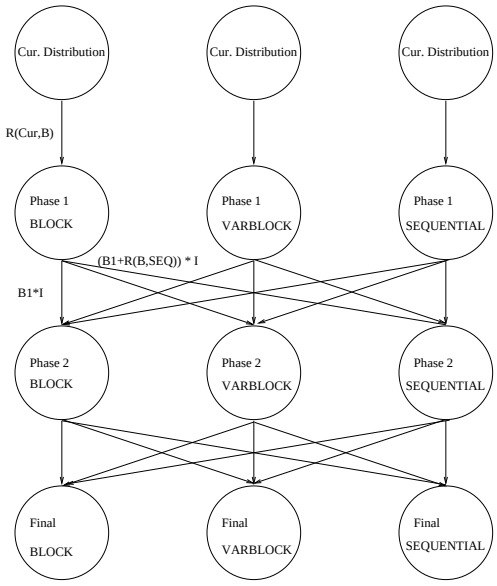

Figure 4: Improved RDDG for the flame simulation. In this figure I denotes the number of remaining iterations, and $B1$ denotes the time to perform phase 1 using `BLOCK`. $R(Cur, B)$ denotes the time to redistribute from the current distribution to `BLOCK`, and $R(B, SEQ)$ from `BLOCK` to sequential. Again, most of the edge weights are omitted and the shortest path through this graph is the best global distribution.

the current distribution. Third, speedup in a phase can sometimes be so poor that it is best to sequentialize the phase. We add a sequential node per phase to our RDDG as well as appropriate edges to consider this possibility

4 Performance

This section reports the performance of our SUIF-Adapt system on four programs: Jacobi iteration, LU decomposition, particle simulation, and flame simulation. The latter two require run-time support to achieve efficient execution. For each application we developed a program using our SUIF-Adapt system. For an accurate comparison, we also developed programs that use a statically chosen (*hand-coded*) distribution. They use only the Filaments DSM, not SUIF-Adapt; this isolates the difference between the SUIF-Adapt program and the hand-coded program to only the choice of data distribution. In our experiments, it is this difference that we are interested in; absolute speedups of all SUIF-Adapt *and* hand-coded programs are relatively poor. This is due to current limitations of the quality of Filaments code that our code generator produces. Below, we present the results of runs on 1, 2, 4, and 8 200 MHz Pentium Pros connected by a 100Mbs Fast Ethernet. The cumulative cost of a page fault, page request, and page transfer on our cluster is 1.7 milliseconds; about 1/3 of this time is overhead within the DSM system.

4.1 Static Applications

The execution times for the two single-distribution applications are shown in Figure 5. Each is statically analyzable; the compiler chooses `BLOCK` for Jacobi iteration (nearest-neighbor communication, uniform workload) and `CYCLIC` for LU decomposition (broadcast communication, decreasing workload). As a result, both SUIF-Adapt programs are competitive with their hand-coded counterparts. Note that in these cases it is possible for the compiler to generate code that avoids any monitoring at all, as a good static distribution is precisely determined. However, we do not do this, because in general we want our system to be able to adapt to load that could potentially be present in a multiuser environment.

4.2 Adaptive Applications

Our particle simulation is based on MP3D from the Splash suite [PWG91]; results are shown in Figure 5. There are three phases and the initial particle positions were evenly divided throughout the grid. However, the particles tend to collide and move toward the upper region of the grid. Although the hand-coded program uses a `BLOCK` distribution to initially balance the load, when the particles move, the load becomes unbalanced. On the other hand, the SUIF-Adapt program performs periodic redistributions over the course of the simulation to rebalance the load and hence performs much better. The speedup tapers off at eight nodes because a perfect balance of the number of particles requires the distribution of *partial* rows. However, distributing partial rows would cause thrashing in our underlying DSM system. Instead, our minimum unit of distribution is a single row.

Our version of flame simulation resembles the code fragment given in Figure 2. The program contains two phases; the first performs a simple nearest-neighbor calculation, and the second is dependent only on local points. Each has a different best local distribution. Both phases contain parameterizable delay loops so that we could experiment with differing phase execution times.

Figure 6 shows the performance of our modified flame simulation. We ran tests with three different delay loops; all were such that in the second phase, the work was concentrated in the top quarter of the grid. The first test weighted the second phase heavily, while the other 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 first two tests were variable block in both phases (first test) and block in both phases (second test). The best distribution for the third test depends on the number of nodes used. With two and four nodes, it is more effective to choose block/variable block with redistribution; this avoids having one node do all the work in the second phase. However, on eight nodes it is better to choose block in both phases, as (1) *two* nodes actually have work to do in the second phase (because the work is concentrated in the top quarter of the grid), and (2) the cost of redistribution when using eight nodes is higher than when using two and four nodes.

For all tests, the run-time analysis of Adapt successfully found these distributions with small overhead compared to the *best version* of the hand-coded programs—that is, Adapt performs comparably against programs coded using *prior knowledge*, which the programmer or compiler does not normally have. For example, it is difficult in general for a programmer or compiler to pick one distribution for two and four nodes and another for eight nodes.

Finally, Figure 7 shows the potential benefit of using the number of remaining loop iterations in a phase cy-

Number of Nodes	1	2	4	8
Jacobi: SUIF-Adapt Time (sec)	226	144	97.5	78.2
Jacobi: Hand-coded Time, `BLOCK` (sec)	226	142	97.1	76.9
LU: SUIF-Adapt Time (sec)	192	110	66.1	53.7
LU: Hand-coded Time, `CYCLIC` (sec)	192	109	65.7	52.7
Particle Simulation: SUIF-Adapt Time (sec)	201	134	89.0	77.5
Particle Simulation: Hand-coded Time, `BLOCK` (sec)	201	153	129	132

Figure 5: Performance of Jacobi iteration, LU Decomposition, and particle simulation. Jacobi iteration uses an 800×800 grid for 500 iterations. LU decomposition uses an 800×800 grid. Particle simulation performs 40 time steps using a grid of 64×64 with 128 particles.

Number of Nodes	1	2	4	8
Flame (1): SUIF-Adapt Time (sec)	206	144	97.6	77.1
Flame (1): BB/BV/VV (sec)	206	198/265/142	181/268/97.2	96.3/273/76.9
Flame (2): SUIF-Adapt Time (sec)	211	137	86.6	53.1
Flame (2): BB/BV/VV (sec)	211	137/341/198	86.0/345/178	51.8/350/169
Flame (3): SUIF-Adapt Time (sec)	218	138	85.9	49.2
Flame (3): BB/BV/VV (sec)	218	141/138/196	92.2/85.8/173	48.6/63.0/162

Figure 6: Performance of 3 versions of our flame simulation, size 1024×1024. 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.

cle along with the current distribution. It is a test of flame simulation (1) on eight nodes for just five iterations. Although the best distribution is to use variable sized blocks in each phase, a global redistribution would be necessary to achieve this. Because there are so few iterations remaining, it is more time-efficient to leave the data distributed in the initial distribution (`BLOCK`).

5 Discussion and Current Research

We view our work as encouraging, but only a first step; currently, our system supports applications where loop bounds need to be shifted. Our current research is aimed at supporting two additional types of applications.

First, we are investigating applications, such as ADI integration and implicit hydrodynamics codes, for which data dependencies require either a complete data reorganization or *software pipelining* [PW86]. While pipelining can often give better performance than transposition in explicit message passing programs, it will almost always cause either thrashing or unnecessary two-way communication in current DSM systems. In separate work we have integrated support for pipelining into DSM systems, so we are well positioned to handle this problem [BL99]. Furthermore, we hope to augment SUIF-Adapt to compare transposition to pipelining on the fly and choose the more efficient one. Finally, if pipelining is chosen, we plan to extend SUIF-Adapt to find the most effective block size [LJ99].

Second, we are examining extensions to our model that will allow dynamic sequentialization *between* phase cycles. Such extensions would allow more efficient execution of multigrid methods, where (1) phase cycles are themselves executed within a loop, and (2) a smaller data set is accessed on each such iteration. At some point, the cost of executing phases within a phase cycle in parallel outweighs the benefit. We would like to be able to dynamically sequentialize phase cycles when profitable. We currently assume that phase cycles are outermost loops. Instead, we will need to allow phase cycles to be nested within other (sequential) loops; then, we will make phase cycle sequentialization decisions after one complete execution of the enclosing loop.

6 Related Work

Compiler techniques to distribute data *within* phases have been studied extensively (e.g., [LC90, BFKK91, O'B93, GB93, RN95]). To find global distributions at compile time, Kennedy and Kremer [KK98] use a "decomposition graph", which was similar to the "communication graph" originally described by Anderson

Number of Nodes	8
Flame(2) [5 iterations]: SUIF-Adapt Time without redistribution (sec)	13.2
Flame(2) [5 iterations]: Time with redistribution (sec)	31.2

Figure 7: Performance of flame simulation for 5 iterations, size 1024×1024, using 8 nodes. SUIF-Adapt takes into account the current distribution and the number of remaining iterations. It does not redistribute, and the savings in time is significant.

and Lam [AL93]. Our RDDG is based on both of these approaches. Others that take a similar approach include [PB95, GAL96]. Unfortunately, if even one phase is unanalyzable, the compiler cannot infer an effective global distribution.

Approaches employing a run-time system, such as ALEXI [Who91], CHAOS [HMS+95], AppLeS [SWB97], and Adapt [LA96], can find an efficient local data distribution even in cases where workload and communication characteristics of a program change at run time. However, a pure run-time system does not necessarily know when phases or phase cycles begin or end, and hence might be unable to choose an appropriate point in the program for data redistribution. Our SUIF-Adapt system tightly integrates a compiler with a run-time system and so can efficiently solve the global data distribution problem.

One project similar to ours has been undertaken by Ioannidis and Dwarkadas [ID98]; they are also investigating balancing load and minimizing communication in DSM systems. However, their work primarily concerns local (single phase or two adjacent phases) data distribution. Finally, many have studied integrated compiler/DSM systems with a focus on elimination of as many consistency actions as possible using compiler information. [KT97, LCD+97, CDLZ97].

7 Summary

We have described the design and implementation of an integrated compiler/run-time system for global data distribution in distributed shared memory (DSM) systems. The SUIF-Adapt system efficiently supports a larger class of applications than previous compiler-only approaches. The compiler parallelizes the code where possible, divides the program into phases, chooses an initial data distribution for each, and inserts calls to the run-time system at appropriate points in the program. The run-time system monitors program execution and finds effective local and global data distributions such that completion time in our framework is minimized. Our SUIF-Adapt system is flexible, as it determines effective distributions even when the phases of an application exhibit vastly different and statically unanalyzable computational characteristics. Performance of our system on several applications was always within 7% of the program that used the best statically determined distribution and outperformed static distributions when phase behavior changes. Further, many of the statically chosen distributions used *prior knowledge* generally unavailable at compile time.

References

[LA96] 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.

[O'B93] Michael O'Boyle. A data partioning algorithm for distributed memory compilation. Technical Report UMCS-93-7-1, University of Manchester, Computer Science Department, July 1993.

[RN95] J. Ramanujam and A. Narayan. Automatic data mapping and program transformations. In *Workshop on Automatic Data Layout and Performance Prediction*, June 1995.

[Who91] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[HAA+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[HKT92] 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.

[FHK+90] Geoffrey Fox, Seema Hirandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Rice University, December 1990.

[LFA96] 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.

[GB92] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techiniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, pages 179–193, 3 1992.

[Low98] David K. Lowenthal. Local and global data distribution in the Filaments package. In *Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 33–41, July 1998.

[KK98] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[PWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Department of Electrical Engineering and Computer Science, Stanford University, April 1991.

[PW86] David Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 12 1986.

[BL99] Karthikeyan Balasubramanian and David K. Lowenthal. Efficient support for pipelining in distributed shared memory systems (submitted to Parallel and Distributed Computing Practices). August 1999.

[LJ99] David K. Lowenthal and Michael James. Run-time selection of block size in pipelined parallel programs. In *Proceedings of the 2nd Merged IPPS/SPDP*, pages 82–87, April 1999.

[LC90] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 424–432, October 1990.

[BFKK91] 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.

[GB93] 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.

[AL93] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, pages 112–125, June 1993.

[PB95] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[GAL96] Jordi Garcia, Eduard Ayguade, and Jesus Labarta. Dynamic data distribution with control flow analysis. In *Supercomputing '96*, November 1996.

[HMS+95] 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.

[SWB97] Gary Shao, Rich Wolski, and Fran Berman. Modeling the cost of redistribu-

tion in scheduling. In *Eighth SIAM Conference on Parallel Processing for Scientific Computation*, March 1997.

[ID98] Sotiris Ioannidis and Sandhya Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-Time Systems for Parallel Computing*, pages 107–122, May 1998.

[KT97] Pete Keleher and Chau-Wen Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[LCD+97] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and distributed shared memory support for irregular applications. In *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–56, June 1997.

[CDLZ97] Alan L. Cox, Sandhya Dwarkadus, Honghui Lu, and Willy Zwanapoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proc. of the 11th International Parallel Processing Symposium*, pages 474–482, April 1997.