Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing

David K. Lowenthal
Vincent W. Freeh
Gregory R. Andrews

Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing

David K. Lowenthal Vincent W. Freeh and Gregory R. Andrews

TR 95-14

Abstract

Programming distributed-memory multiprocessors and networks of workstations requires deciding what can execute concurrently, how processes communicate, and where data is placed. These decisions can be made statically by a programmer or compiler, or they can be made dynamically at run time. Using run-time decisions leads to a simpler interface—because decisions are implicit—and it can lead to better decisions—because more information is available. This paper examines the costs, benefits, and details of making decisions at run time. The starting point is explicit fine-grain parallelism with any number (even thousands) of threads. Five specific techniques are considered: (1) implicitly coarsening the granularity of parallelism, (2) using implicit communication implemented by a distributed shared memory, (3) overlapping computation and communication, (4) adaptively moving threads and data between nodes to minimize communication and balance load, and (5) dynamically remapping data to pages to avoid false sharing. Details are given on the performance of each of these techniques as well as their overall performance on several scientific applications.

December 1, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[0]This research was supported by NSF grants CCR-9415303 and CDA-8822652.

1 Introduction

Writing efficient parallel programs for distributed-memory multiprocessors and networks of workstations requires addressing five issues:

- discovering parallelism,
- determining the granularity of parallelism,
- exchanging data between processors,
- overlapping communication and computation, and
- distributing data among nodes.

The first issue has to be addressed when a program is developed or compiled, because parallelism cannot be discovered at run time. However, the other four issues can be addressed at run time. If an issue is addressed by a programmer or compiler, the solution is static. On the other hand, run-time solutions are dynamic.

This paper describes and compares static and dynamic approaches to addressing the issues of granularity, communication, overlap, and data placement. We show that making decisions at run time simplifies application programs and compilers (because decisions are made implicitly), that run-time decisions can be implemented efficiently, and that they can result in improved overall performance (because more information is available than at compile time).

To be more precise, a completely static program for a distributed-memory multiprocessor typically specifies some small number of coarse-grain processes (often one per processor), uses message passing for communication between processors, and explicitly places the data. We are exploring the other extreme: We start with fine-grain parallelism and then address the other issues dynamically.

Our approach presents both challenges and opportunities. The challenges are to efficiently implement fine-grain concurrency, run-time coarsening of fine-grain processes, implicit communication, and adaptive data placement. The opportunities are to overlap computation and communication and to make better decisions. Moreover, this approach provides a simpler interface for application programmers and compilers, because the run-time system solves most of the difficult problems, freeing the application programmer and compiler writer to concentrate on other areas.

We have implemented our approach in an experimental software system called Filaments [FLA94]. The relevant attributes of Filaments are:

- fine-grain parallelism with any numbers of threads, even hundreds of thousands;
- implicit communication by means of a distributed shared memory (DSM);
- implicitly overlapped communication and computation;
- dynamic movement of threads and data among nodes to minimize communication and balance the load; and
- dynamic remapping of data to pages to avoid false sharing.

Using Filaments, the application programmer or compiler has to decide what parts of the program can be executed in parallel, *but that is all.* The programmer or compiler can ignore the issues of the granularity of parallelism, how processes communicate, and so on. Consequently, Filaments programs are portable between vastly different machines; in particular, the Filaments programs used in this paper run efficiently on both shared-machine multiprocessors and a cluster of workstations.

The organization of the paper is as follows. The next section more fully describes the static and dynamic (run-time) approaches using a representative application (Jacobi iteration); it also summarizes the issues involved in making the run-time approach efficient. Section 3 describes the implementation and performance of the concurrency and communication mechanisms provided by Filaments. Section 4 describes the implementation and performance of our run-time mechanisms for moving data between nodes and pages. Section 5 analyzes the performance of each mechanism for Jacobi iteration and their overall performance for Jacobi and three additional applications. Finally, Section 6 gives concluding remarks. Although all topics are discussed in the context of their implementation in the Filaments package, the techniques and conclusions apply to other systems for fine-grain parallelism or distributed shared memory.

2 Overview of the Static and Run-Time Approaches

Our focus is on writing and executing iterative scientific applications for distributed memory machines. This section describes one such application, Jacobi iteration; shows how it would typically be written as a coarse-grain program with explicit communication and a statically determined data placement; shows how it could be written as a fine-grain program in Filaments with implicit communication and adaptive data placement; and then summarizes both the challenges and opportunities presented by the latter in order to make it run efficiently.

2.1 An Example of a Static Approach

Laplace's equation in two dimensions is the partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

Given boundary values for a region, its solution is the steady values of interior points. These values can be approximated numerically by using a finite difference method such as Jacobi iteration. In particular, discretize the region using a grid of equally spaced points, and initialize each point to some value. Then repeatedly compute a new value for each grid point; the new value for a point is the average of the values of its four neighbors from the previous iteration. The computation terminates when all new values are within some tolerance, `EPSILON`, of all old values.

Assume there are P nodes on a distributed-memory machine. To implement Jacobi iteration as a coarse-grain program with explicit communication and a statically determined data placement, we do the following. First, we use one process per node. Second, we need to distribute the grids of old and new values among the nodes. A natural and efficient way to do this is to assign each process a *strip* of each grid—a contiguous set of n/P rows. These become local arrays within each process. (A strip assignment of data to nodes maximizes the locality within each process and minimizes the number of "edges" between processes.)

A process "owns" the points in its strip; i.e., it is the only process that can read and write those values. Each process repeatedly executes six phases: send the top and bottom rows of new values to neighbors, update local grid points, receive the top and bottom rows of new values from neighbors, update boundary points, swap the roles of the old and new arrays, and check for termination. The processes need to exchange their top and bottom rows because the new values for points in these rows depend on old values computed by the processes that own the adjacent rows. By asynchronously sending the values first and receiving them after performing local work, the program overlaps communication and computation.

The processes also need to interact to detect termination. Each can determine whether all new values in its strip are within EPSILON of all old values, but these local decisions need to be combined to determine whether the computation as a whole should terminate. For simplicity, we show a central coordinator process to detect termination. The process that starts the computation acts as the coordinator. (One of the computational processes could assume the role of the coordinator.)

An outline of the code for each computational process follows. The parameter pid is the unique identifier of each process.

```
void jacobi(int pid) {

  /*  declarations of matrices for strips of old and new grids,
   *  startrow, endrow, localdiff, etc.
   *
   *  initialize local variables                                    */

  startrow = pid * n/P;
  endrow = startrow + n/P;
  init_grids();   /* receive initial values for grid or compute locally */

  while (!done) {
    send_rows();      /* send top and bottom rows to neighbor processes */
    /* compute interior */
    for (i = startrow+1; i < endrow-1; i++) {
      for (j = 1; j < n; j++) {
        new[i][j] = (old[i+1][j]+other neighbors)/4.0;
        temp = absval(new[i][j] - old[i][j]);
        if (temp > localdiff)
          localdiff = temp;
      }
    }
    recv_rows();        /* receive top and bottom rows from neighbors */
    /* compute boundaries; code is similar to that for interior points */

    send_diff();            /* send localdiff to coordinator process */
    swap(old, new);
    done = recv_done();     /* receive termination result */
    if (done)
      send_grid();          /* send local strip to coordinator */
  }
}
```

The exchanging of messages with the coordinator effectively introduces a barrier synchronization point. In particular, none of the processes proceeds to a new update phase until the coordinator has gathered the results of the previous phase from each process and sent replies.

The coordinator process initiates the computation, receives local differences, and sends replies to the jacobi processes. An outline of its actions is:

```
void coordinator() {
  /* declarations of local variables (maxdiff, etc.) */
  /* create P instances of jacobi() */
  /* send initial data to each process if it isn't computed locally*/
  init_grids();

  while (!done) {
    maxdiff = recv_diffs(); /* receive a localdiff from each process */
    if (maxdiff < EPSILON)
      done = true;
    broadcast_done(done);   /* send the vaue of done to each process */
  }
  recv_grids();             /* receive results from each process */
}
```

2.2 An Example of a Run-Time Approach Using Filaments

The Filaments package [FLA94, EAL93] supports fine-grain parallelism and a shared-memory programming model on the entire range of parallel machines, from shared-memory multiprocessors to networks of workstations. In this paper we limit our discussion to networks of uniprocessor workstations, which we refer to as *nodes*. A *filament* is a very lightweight thread that can access the shared memory. There are two kinds of filaments: iterative and fork/join. We focus on iterative filaments, which execute repeatedly, with barrier synchronization and termination detection occurring after each execution of all filaments.

A program that uses the Filaments package has three additional components relative to a sequential program:

- declaration and allocation of variables that are to be located in shared memory;
- code executed by individual filaments; and
- a section that initializes the package, creates the filaments, places them on nodes, and times and controls their execution.

Filaments are executed by *server threads*, which are conventional lightweight threads with a stack and context (unlike a filament, which has no private stack or context). Each node has at least one server thread. Each filament is placed in a *pool*, which is a group of filaments that ideally have a similar data-reference pattern. The collection of pools on a node is called a *pool set*; it also has an associated function that is called after each execution of all filaments in the pools in the set; this function most often synchronizes the nodes and checks for termination.

We now describe how to implement Jacobi iteration using the Filaments package. For this application, the shared variables are the two grids. The code executed by each filament computes an average and difference. Because Jacobi iteration uses two grids, the n^2 updates are all independent computations; hence, all new values can be computed in parallel. The initialization section sets up the Filaments package and the matrices, creates the pool sets, pools, and filaments, and then starts the server threads on each node.

The important shared variables are:

```
double **new, **old;
fReductionVar(double, maxdiff);  /* a reduction variable */
```

The new and old variables are dynamically allocated two-dimensional vectors of matrices. (The boundaries of the region are stored in the edges of the arrays.) The fReductionVar macro declares a *reduction* variable, which is a special kind of variable with one copy per node. Here, maxdiff is of type double and is shared among all filaments on a node. The local copy of a reduction variable can be accessed directly. In addition, such variables are used in calls to fReduce, which atomically combines all node's private copies into a single copy using an associative/commutative operator such as add or minimum. Thus, a call of fReduce also results in a barrier synchronization.

Procedure jacobi contains the code executed by each filament:

```
void jacobi(int i, j) {
  double temp;
  new[i][j] = (old[i-1][j] + other neighbors) * 0.25;
  temp = absval(new[i][j] - old[i][j]);
  if (temp > maxdiff)
    maxdiff = temp;
}
```

After computing the new value of grid point (i,j), jacobi computes the difference between the old and new values of that point. If the difference is larger than the maximum difference seen on this iteration of the entire computation, then maxdiff needs to be updated.

After all grid points are updated, the following procedure is called to check for convergence and to swap grids:

```
int termCode() {
  fReduce(maxdiff, MAX);
  if (maxdiff < EPSILON)
    return DONE;
  swap(old, new);
  maxdiff = 0.0;
  return CONTINUE;
}
```

This code is executed sequentially on all nodes at the end of every update phase, i.e., after every filament completes its work. The call to fReduce combines all copies of maxdiff (using the MAX operator) into a consistent copy seen by all nodes.

The main procedure, which includes the initialization section, is shown below. A single thread on each node executes this code. The Filaments shared-memory abstraction is implemented by a distributed shared memory (DSM). The two most important parts of this section are the placement of the data on pages of shared memory and the distribution of DSM pages among the nodes. (Because of the shared-memory abstraction, data can be accessed by any node, so the placement of filaments indirectly determines the placement of shared data pages.) The call of fStart executes the filaments that have been created, terminating when all have completed.

```
void main() {
  /* declarations of local variables */

  /* create distributed shared memory and
   * place variables new and old onto pages of shared memory
   * (don't worry about thrashing; will be corrected while
```

```
   * program is executing if necessary) */

  /* create a pool set and some number of pools */
  ps = fPoolSet(2, jacobi, termCode);
  for ( i = 0; i < num_pools; i++ )
    p[i] = fPool(ps, number_of_filaments);

  /* Compute startrow and endrow, which specify which rows of
   * filaments are to be placed on each node
   * (can be arbitrary; best placement found while program executing) */

  /* create the filaments */
  for (i = startrow; i < endrow; i++) {
    pool = whichPool();     /* determine which pool to use for this row */
    for (j = 1; j < n; j++) {
      fCreateFilament(*ps, p[pool], i, j);
    }
  }
  fStart();
}
```

The code above divides filaments into pools based on their data reference patterns. Filaments in the top and bottom rows on each node can potentially reference off-node data, so they are placed in their own pools. All other filaments are placed in a third pool. (Section 3 discusses pools in more detail.)

2.3 Comparison, Challenges, and Opportunities

The code outlines presented above for the static and run-time approaches are about the same length. However, there are several differences, and these make the Filaments program easier to write and hence to understand than the static program. First, it is not necessary to program the clustering (coarsening) of parallelism. Second, the Filaments code directly accesses shared variables instead of passing data in messages; this also makes it easy to provide and use special synchronization variables such as reductions. Third, the Filaments program does not have to worry about overlapping communication and computation, because this is handled by the run-time system, as described in the next section. Finally, the initial placement of data is not critical, because a possibly poor placement will be corrected at run time by Filaments subsystems described in Section 4.

By contrast, the static program clusters points into strips, uses message passing to exchange rows, and intersperses computation with the message exchange to achieve overlap. All of these have to be programmed explicitly, and the code is intermingled to an extent. The static program must also explicitly place the data on the nodes; a poor placement will result in poor performance. The initialization section in the fine-grain program *appears* to be more complex than that of the coarse-grain program, but this is because we have shown all the details of initializing the Filaments package. In reality, the coarse-grain program would also have to initialize the system-call library that it uses (such as PVM).

The Filaments program presents both challenges and opportunities. The challenges are to implement filaments, the shared-memory abstraction, and run-time placement of filaments and data efficiently. The opportunities are to overlap computation and communication by taking advantage of the wealth of concurrency, and to customize the placement of filaments and data at runtime by adapting to the actual data reference pattern of an application. The remainder of the paper describes how these challenges can be overcome and how these opportunities can be realized.

3 Fine-Grain Parallelism and Implicit Communication

This section describes how the Filaments package implements fine-grain concurrency and a distributed shared memory and then describes application-independent performance measurements. Although the discussion focuses on attributes of Filaments, the issues, solution techniques, and conclusions apply to similar packages.

A filament consists only of a code pointer and arguments; it does not have a private stack. The filaments in a program communicate by referencing shared variables. The shared memory abstraction is implemented by a distributed shared memory (DSM) that is customized for use with fine-grain threads. The shared address space is divided into pages; copies of pages move between nodes and may be replicated. When a filament references a location on a page that is not local, a page fault occurs.

Filaments are executed one at a time by *server threads*, which are conventional lightweight threads with stacks. Each node starts with one server thread and creates others as needed. In particular, filaments are placed in *pools* based on their data-reference patterns; each node has one or more pools. When a server thread executes a filament that causes a page fault, the server thread is suspended. If there is another pool of filaments on that node, another server thread is created and begins executing filaments in that pool. This allows computation to be overlapped with communication.

3.1 Implementing Fine-Grain Threads (Filaments)

A pool is represented by a list (array) of filaments and a pointer to a function. The basic execution model of Filaments is to have a server thread traverse the list, calling the pool's function with arguments specified by the filament itself. (In fact, one can view a filament as simply a set of arguments.) Implementing this execution model efficiently depends on controlling the overheads involved in executing many small filaments. Specifically, the overheads are creating and running filaments, potentially inefficient use of the cache, and producing code that can be hard for a compiler to optimize because of the abundance of function calls and pointers.

Many Filaments programs attain good performance with little or no optimization (e.g., matrix multiplication). In such applications, each filament performs a significant amount of work ($O(n)$ in matrix multiplication), which amortizes the filament overheads. However, achieving good performance for iterative applications that possess many small filaments (e.g., Jacobi iteration, where each filament only performs a few instructions) requires using *implicit coarsening*. In particular, filaments in a pool are executed as if the application were written as a coarse-grain program.[1] To implement implicit coarsening, we use two techniques: inlining and pattern recognition. These reduce the cost of running filaments, reduce the working set size to make more efficient use of the cache, and use code that is amenable to compiler optimizations.

Inlining, as the term implies, consists of inlining the body of each filament rather than making a procedure call. In particular, when processing a pool, a server thread executes a loop, the body of which is the code specified by filaments in the pool. This eliminates a function call for each

[1] Systems such as Chores [EZ93] and the Uniform System [TC88] have a fine-grain specification and a coarse-grain execution model, but use preprocessor support. Filaments generates different codes at compile time, but chooses among them at run time.

filament, but the server thread still has to traverse the list of filament descriptors and load the arguments.

The second technique is to recognize common patterns of filaments at run-time. Filaments recognizes regular patterns of filaments assigned to the same pool. In such cases, the package *at run time* switches to code that iterates over the filaments, generating the arguments in registers rather than reading the filament descriptors. Filaments currently recognizes a few common patterns that support a large subset of regular problems; however, the technique is capable of supporting any number of other patterns.

Efficiently implementing many small filaments also requires avoiding excessive faulting. In an application that creates many filaments, it is likely that several of these filaments will reference data on the same page, potentially causing many faults on this page. This issue is addressed by using pools. The application places filaments with similar data access patterns into a pool on a node at initialization time. When a program is started, a server thread on each node starts executing one pool of filaments. This thread executes pools of filaments until either a page fault occurs or all eligible filaments have been executed. On a page fault, the state of the filament is saved on the stack of its server thread, and a new server thread is started; it executes filaments in a different pool while the remote page is being fetched. Thus, an *entire* pool is suspended when any one of its filaments faults. This minimizes page faults if filaments in the same pool reference the same pages.

In addition to avoiding excessive faulting, the pool mechanism also is responsible for realizing one of the key opportunities available in implementing fine-grain threads: overlap of communication and computation. The next subsection provides details of how pools help to achieve maximal overlap.

The application program determines the number of pools it should use on each node and assigns each filament to a pool and a node when the filament is created. Section 4.1 describes an adaptive algorithm that determines how to place filaments on nodes at run time, allowing the application program to effectively ignore the issue. If a node has a single pool, it is essentially single-threaded; this would work well if the time to switch context to a new server thread is greater than the time to fetch a remote page. Normally, however, the application will want to use multiple pools—as we did in the application in Section 2.2—as this increases the possibility of overlapping computation and communication. We are currently working on an adaptive algorithm for determining the number of pools and assigning filaments to them.

3.2 Distributed Shared Memory

Our multi-threaded distributed shared memory (DSM) is implemented entirely in software and therefore requires no specialized hardware or changes to the operating system kernel. In single-threaded DSM implementations, such as [FP89, CBZ91, KDCZ94, SFL$^+$94, BZS93, DJAR91, BKT90], all work on a faulting node is suspended until the fault is handled. In a multi-threaded implementation, other work is done while the remote fault is pending. This makes it possible to overlap communication and computation. VISA, a DSM written for the functional language Sisal, allows less general overlap of communication and computation [HB92].

The address space of each node contains both shared and private sections. Shared user data (matrices, linked lists, etc.) are stored in the shared section, which is divided into individually protected pages of 4K bytes each. Local user data (program code, loop variables, etc.) and all system data structures (queues, page tables, etc.) are stored in the private sections. The shared section is replicated on all nodes in the same location so that pointers into the shared space have the same meaning on all nodes.

Two key events occur in our DSM system: *remote page fault* and *message pending*. A remote page fault is generated when a server thread tries to access a remote memory location. It is handled

Operation	Time (μs)	ops/sec
Filaments creation	2.10	457,000
Context switch		
Filaments	0.643	1,560,000
Fil. Inlined	0.126	7,950,000
Threads	48.8	20,500

Figure 1: Filaments overheads

by using the `mprotect` system call, which changes the access permission of pages, and by using a signal handler for segmentation violations. A message pending event is generated when a message arrives at a node; it is handled by an asynchronous event handler, which is triggered by the I/O pending interrupt (`SIGIO`).

When a filament accesses a remote page, the server thread executing the filament is interrupted by a signal. The signal handler inserts the faulted server thread in the suspended queue for that page, requests the remote page if necessary, and calls the scheduler, which will execute another server thread. When the request is satisfied, the faulted server thread is rescheduled, as are all other server threads that are waiting on that page. Because a new server thread is run after every page fault, the system can have several outstanding page requests.

Filaments uses the multi-threaded DSM together with the pool mechanism to achieve maximal overlap of communication and computation. For iterative applications, Filaments ensures that after the first iteration, the pools that are run first will be those that faulted on the previous iteration; as many iterative applications have constant sharing patterns, these pools will likely fault again. This "front loads" the page faults, which increases the potential for overlapping communication and computation, because there is the maximum amount of local work to do while the faults are being satisfied.

Filaments implements the frontloading of page faults in the following way. On a page reply, the enabled server threads are placed on the tail of the ready queue. This ensures that pools containing at least one filament that faults will finish execution after a pool that contains no filaments that fault, provided that the faulting pool is started before the non-faulting pool. (Non-faulting pools are always run to completion, because server threads are only suspended on a page fault.) To make sure all faulting pools are started first, when a server thread finishes executing an entire pool of filaments, it pushes the pool on a stack. On the next iteration, the pools are run starting at the top of the stack, which ensures that all faulting pools are run first.[2]

A DSM has to implement one or more page consistency protocols (PCPs). We have implemented several, including write-invalidate [LH89], implicit invalidate [FLA94], write-shared [CBZ91], and writer-owns [Fre95]. (PCPs are discussed in greater detail in Section 4.2.) A DSM also requires reliable communication. Our system uses a novel, low overhead reliable datagram subsystem called Packet, which is beyond the scope of this paper (see [FLA94]).

3.3 Application-Independent Overheads

The performance of Filaments programs are application-dependent. For example, an application with a large ratio of computation to communication will perform much better than one with a small ratio. This is because the communication overheads in the Filaments DSM are better amortized

[2]We have not found iterative applications that possess a sharing pattern for which this algorithm is not optimal. However, if such an application does exist, we can front load the faults by running one filament from each pool at the beginning of each iteration.

Action	Time (μsec)
Send Message	41
Receive Message	424
Make Page Request	967
Service Page Request	1642

Figure 2: Cost of various Filaments communication and overlap operations

Nodes	2	4	8
Time (msec)	3.20	5.29	8.45

Figure 3: Barrier synchronization times

by having a lot of computation relative to communication. However, there are inherent overheads in Filaments that are independent of any particular application.

Some of the filaments overheads are shown in Figure 1. Each is shown both as the time per operation and as the number of operations per second. The cost of switching between filaments depends on whether or not they are inlined. Inlining filaments eliminates a function call (and return) and improves performance by more than five-fold. For comparison purposes, context switch times for the lightweight server threads are shown as well.

Another main Filaments overhead is due to DSM paging. There are four costs associated with DSM paging: faulting on the page, and sending, receiving and servicing the message. The faulting node incurs the first three overheads and the owner of the requested page bears the latter. The paging overhead is application dependent. In general it does not depend on the number of nodes, but on the sharing of data. Quite often the number of messages increases linearly with the number of nodes, which only becomes a problem when the network is saturated.

The final Filaments overhead is due to synchronization, which results from barriers in iterative applications. The overhead of barriers is a function of the number of nodes. Filaments uses a tournament barrier with broadcast dissemination, which has $O(p)$ messages and a latency of $O(\log p)$ messages [HFM88]. Barrier synchronization times are shown in Figure 3. This is the cost of the barrier only; in an actual application it is likely that the nodes arrive at the barrier at different times, which increases the time a particular node is at the barrier.

4 Data Placement

With a DSM, any node can reference any variable. However, variables are placed on pages, and if a page is not resident, referencing a variable on that page leads to a page fault. Moreover, a page usually contains more than one variable, such as elements of an array. This can lead to false sharing, which occurs when different nodes update different variables that happen to be located on the same page.

This section considers the problems of placing data on nodes and pages and presents implicit mechanisms for moving data at runtime. The challenges are to minimize communication, balance the computational load, and avoid false sharing (which can lead to thrashing). The ideal data placement minimizes the overall completion time of an application. The mechanisms we describe introduce some overhead due to runtime monitoring, but they also make it possible to adapt dynamically to the characteristics of an application. This sometimes leads to better performance than is possible using any static choice for data placement. Below we give an overview of these systems; their performance is discussed in Section 5.

4.1 Adaptive Placement of Data on Nodes

First consider the problem of placing data (pages) on nodes. Most current approaches determine data placements statically. They can generally be divided into two categories: using language primitives, such as the ones in HPF [HPF93], or compiler analysis, such as the work reported in [AL93], [GB93], and [KK94]. Language primitives involve the programmer in the choice of data placement; unfortunately, the best placement may be difficult or impossible for the programmer to determine. Compiler analysis also may not be able to infer the best data placement; moreover, the difficulty of inferring placements greatly increases the size and complexity of the compiler.

Our approach is to determine data placements dynamically, without requiring programmers or compilers to make such decisions. (Different dynamic approaches are discussed in [Who91] and [HMS+95].) This approach is implemented in a prototype system called Adapt [LA95], which is a subsystem of the Filaments package.

The goal of Adapt is to minimize the overall completion time of an application, which is determined by the completion time of the slowest node. Three factors affect the completion time of a node: computation time, communication overhead, and delay. Computation time is the time spent executing application code, communication overhead is time spent executing low-level code that copies messages to and from the network, and delay is time spent waiting for other nodes to complete their computation or respond to a message. We assume that any node can reference any data element. We also assume the *owner-computes* rule [HKT92]. This means each data element has an "owner", which is the only node that updates the element; however, other nodes may reference the element.

The elements of a data structure can be placed on the nodes in numerous ways. However, the challenges of simultaneously balancing computational load and minimizing communication often conflict, as there is an interaction between the two. For example, one placement extreme is to put all data elements on one node; this will minimize communication (there is none), but it also maximizes load imbalance (all other nodes are idle), which leads to large delays at barrier points. The other extreme is to assign elements randomly to nodes; this will (probabilistically) balance the load, but the lack of spatial locality will most likely lead to a large amount of communication.

Between these extremes are several feasible data placements. Adapt considers three—block, variable block, and cyclic—as illustrated in Figure 4. A *block* placement places a logically contiguous set of approximately the same number of data elements on each node. This mapping (called `BLOCK` in HPF) could, for example, place contiguous rows (or columns) of a matrix on each node. Block placements tend to work well for stencil-based applications such as Jacobi iteration, because such applications have spatial and temporal locality, a balanced workload, and regular communication between neighboring nodes. Block placements also work well for applications such as particle-in-cell codes [Har64], that have locality and a regular "nearest neighbor" communication pattern. In this case, however, the sizes of blocks may need to vary in order to balance the workload; e.g., each block should contain about the same number of particles. (There are no variable-block placements in HPF.)

Another placement method is to *stripe* data across the nodes (this is called `CYCLIC` in HPF). Striped placements can handle problems with changing workloads well, because if the amount of work per element decreases within the computation, a striped placement balances the load without a need for remapping. However, striped placements have fairly poor spatial locality, so they are typically useful only when the amount of communication in an application is (relatively)

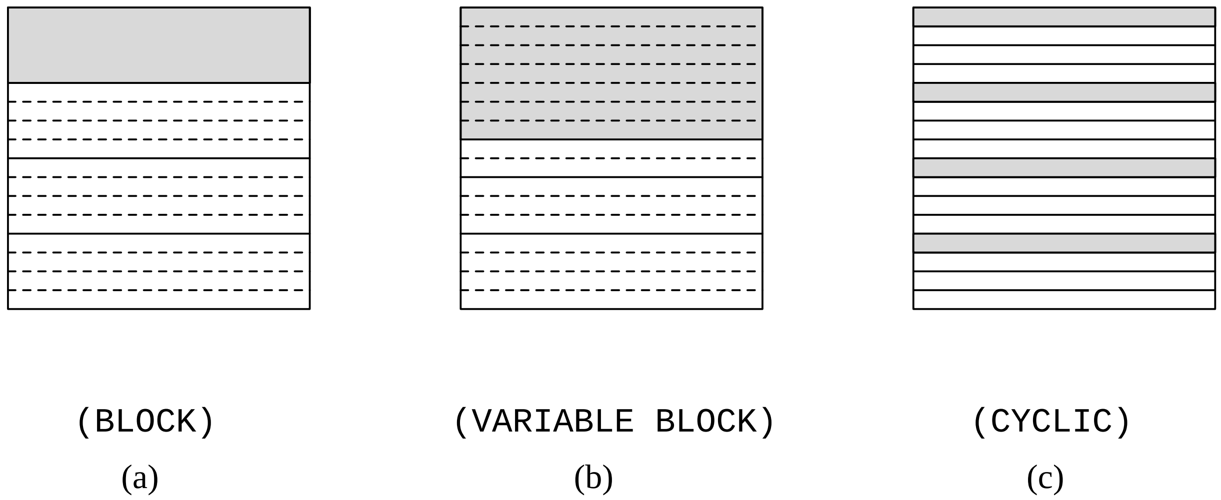

Figure 4: Different data placements in Adapt for the case of 16 rows and 4 nodes. Three different placements are shown: block, one possible variable block, and cyclic. The rows that are placed on node 0 are shaded. Solid lines separate different node's data.

independent of the data placement. LU decomposition is an example of an application with a changing workload and a placement-independent communication pattern. Compromise placements can also be useful, such as striping contiguous regions onto each node (see [LA95] for details).

The Adapt system dynamically selects one of the data placements shown in Figure 4. It is given some initial data placement by the programmer or compiler (the current default is `BLOCK`) and then employs three steps to determine whether this placement is a good one or whether it should be changed. First, Adapt gathers information about the communication pattern and computation time for each loop body in the application. Next, it uses this information to determine which data placement is likely to minimize both communication overhead and delay. Finally, it effects the new placement (if necessary) and continues to monitor the computation in case the amount of computation or communication later changes.

Adapt monitors communication using DSM page faults and the DSM page table. In particular, the system counts the number of messages that each node sends and receives during one iteration of the application program. From these counts—and architecture-specific measures of the times it takes to send pages between nodes and to service page requests—Adapt estimates the time due to communication overhead and message delay on each node. Adapt also determines the communication pattern by inspecting the pattern of page faults on each shared array. (Currently, Adapt recognizes two patterns: *nearest-neighbor* and *broadcast*.)

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

After gathering communication and computation information for one iteration of an application, Adapt uses it to choose a good data placement. 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 is done using a simple bin-packing procedure, which in turn depends on the communication pattern detected during the monitoring phase. When the communication pattern is nearest-neighbor, Adapt packs the bins so that each bin contains *consecutive* rows and the estimated total time on the node is as close as possible to T/P. Adapt also investigates multiple-bin packings if the load is not sufficiently balanced. When the communication pattern is broadcast, the type of packing depends on the workload. If a history of iteration execution times shows a constant workload, the same procedure as the one above is used. On the other hand, if the execution times are changing, Adapt uses n/P bins on each node to effect a `CYCLIC` style placement.

If Adapt determines that a new data placement would be better, it reparameterizes the application (filament) code so that the filaments on each node will access different data. This causes some filaments to reference data that the node does not own, and hence causes page faults. The underlying DSM in Filaments then implicitly moves the data.

Adapt continues to monitor the application to detect when a different placement might be better. A large variance in the computation times suggests an imbalanced load, which might require a placement that better balances the load. An increase in the communication times suggests excess communication, which might require a placement with more locality. If either is detected, Adapt notifies the nodes before the start of the next iteration. All nodes then re-enable the fine-grain monitoring (time each row, etc.) and repeat the basic algorithm to determine the new (if any) best placement.

4.2 Adaptive Placement of Data on Pages

The previous section described the problem of placing pages of data on nodes. We now consider the problem of placing data on pages themselves. The key issue is avoiding false sharing. False sharing occurs when two (or more) nodes are accessing distinct elements of a page and at least one of them is writing.[3] False sharing is a problem, because it can lead to thrashing, a situation in which a page is continuously moving between nodes with very little useful work occurring. This subsection describes the writer-owns protocol [Fre95] that dynamically eliminates false sharing.

A *page-consistency protocol* (PCP) is a method for maintaining data on a page using some memory consistency model [Mos93]. A single-copy PCP, such as migratory [CBZ91], is one in which only one copy of a page ever exists; therefore, it is inherently consistent. In a multiple-reader/single-writer protocol, such as write-invalidate [LH89], a page remains consistent at all times because all read copies are invalidated when a node writes. Multiple-writer protocols, such as write-shared [CBZ91], allow the local copies of pages to become inconsistent; they regain consistency at specific points in the program through some "consistency operation" (e.g., a barrier synchronization). A single-copy PCP is very simple, but limits concurrency because all accesses are serialized. Multiple-copy PCPs are more complicated, but allow greater concurrency.

Suppose false sharing occurs in an application. With the write-invalidate (WI) protocol, any time a write occurs, the read copies on the other nodes will have to be invalidated. If another node has not finished reading the data on the page, it will immediately request a new read copy of the page. This results in thrashing, which can cause considerable message traffic and may limit concurrency, because the readers must wait for the writer and vice versa. In order for the application to achieve adequate performance, thrashing must be avoided or controlled. It can be avoided if the programmer or compiler places data on pages so that no false sharing occurs; this will result in unused space on pages. Alternatively, thrashing can be controlled if the run-time system keeps a page on a node for some minimum period of time (the *time window coherence* protocol of Mirage [FP89] is an example).

The write-shared protocol (WS) tolerates false sharing. Because there can be multiple copies

[3] When nodes are accessing the *same location* with at least one node writing, true sharing occurs. If at least one is updating the element and there is no synchronization between the nodes, there is a race condition, which is a programming problem.

of a page while it is being updated, false sharing does not cause thrashing. However, the page becomes inconsistent (the local copies change), so false sharing forces a consistency operation. In WS consistency is regained by merging local changes. When a node updates a shared page, it saves a (consistent) copy of the original page. At the consistency point the node creates a list of all the changes that were made to the page; this is called a *diff list*. Nodes then exchange and merge diff lists into their copy of the page to regain a consistency.

The writer-owns (WO) protocol detects and eliminates false sharing *at run time*. WO tolerates false sharing when it first occurs in a way similar to WS; however, at the consistency point, the data is relocated to *eliminate* further false sharing.

Avoiding false sharing while using write-invalidate requires detecting and eliminating false sharing at compile time. This is always difficult and sometimes impossible (especially in the presence of pointers). In contrast, writer-owns detects all false sharing as it occurs; hence, detection is simple and exact. The write-shared protocol tolerates false sharing, but differences between pages have to be resolved after *every* iteration of an iterative application. On the other hand, writer-owns tolerates each instance of false sharing only once. Thus WO requires significantly fewer consistency operations in iterative computations than WS.

There are two main steps in implementing the writer-owns protocol. First, during computation, detect when false sharing occurs and tolerate it. Then, at the next consistency point (e.g., barrier), regain consistency and eliminate false sharing by relocating data. Writer-owns requires that the data is "remappable." In particular, because the protocol adjusts pointers, the base *remappable unit* must have a level of indirection. For example, in an n-dimensional array, the rows are the base remappable unit—not the individual elements.

False sharing is detected and tolerated as follows. When an unshared page is requested, the owner of the page sends a copy of the page to the requester and sets the permission of the page to READ_ONLY. When the requester receives the page, it also sets the permission to READ_ONLY. On a subsequent write to the page (by either node) the writing node copies the current copy of the page into a *clone* and changes the permission of the page to READ_WRITE. A page has false sharing if and only if it has been cloned on at least one node.

The second step occurs at the next consistency point. False sharing on a page is eliminated by performing four operations: determining write sets and relocation information, relocating data, disseminating relocation information, and finally remapping the data structures. The clone that was made when false sharing was detected is a replica of the last consistent copy of the page. Therefore, each node can compare the current contents of each shared page to its clone and determine the changes that have been made locally since the last point at which the page was consistent. From this, each node can construct a *write set* that lists all the remappable units (on shared pages only) that the node updated.

In the second step, each node relocates every remappable unit in its write set. The data in each remappable unit move onto pages owned by the node. Consequently, data dynamically migrates into the memory of the node that performs the updates—that is, the writer owns the data.

The third step involves piggy backing *relocation information* on the synchronization message. Relocation information contains the identity of remappable units and their new location. The relocation information from all nodes is collected and disseminated back to all nodes on the acknowledgement to the message.

The last operation remaps the data so that all changes are observed by all nodes. This requires that each node update its pointers to the remappable units that have migrated. It is possible because all data objects are allocated by the WO protocol. In particular, WO provides routines to allocate objects, maintains a data base of objects, and matches locations to remappable units.

In order for the writer-owns protocol to function correctly, the program must observe the *one-*

filament/point (no coarsening)	271
filament/row (no coarsening)	196
filament/node (no coarsening)	192
filament/point (run-time coarsening)	197
Sequential Program	191

Figure 5: Cost of fine-grain parallelism on a single node. (Times in seconds.)

Nodes	1	2	4	8
Coarse-grain, no overlap (sec)	191	98.1	52.6	33.3
Filaments, one per node (sec)	192	98.5	53.8	33.5

Figure 6: Test of cost of implicit communication. (Times in seconds.)

writer rule: only one node may write to a remappable unit in between consistency points. If this rule were violated, both nodes would relocate the same remappable unit. The current system aborts when it detects a violation of the one-writer rule. However, it could instead regain consistency by merging the local changes on the offending remappable unit in a way similar to WS.

In summary, writer-owns does not require static analysis to eliminate false sharing, whereas write-invalidate does. Furthermore, in iterative computations, it amortizes the cost of tolerating false sharing over all iterations and eliminates consistency operations on subsequent iterations; this can lead to better performance than the write-shared protocol.

5 Performance

This section reports the performance of run-time decision making using Filaments. Section 5.1 studies each decision in isolation and then shows the effect of combining several of these decisions; Jacobi iteration is used as the application. The final two sections show results specific to the Adapt and writer-owns.

All tests were run on a isolated network of 8 Sparc-1s connected by a 10Mbs Ethernet. They use the `gcc` compiler with the `-O` flag for optimization. The execution times reported are the median of at least three test runs, as reported by `gettimeofday`.

5.1 Jacobi Iteration

This application was described in Section 2. All Jacobi iteration tests in this section operate on a 512×512 matrix and perform 100 iterations.

Although programming using fine-grain parallelism is often easier than using coarse-grain parallelism, fine-grain parallelism is generally avoided because it is believed to be inefficient. The Filaments package executes fine-grain parallelism efficiently; Figure 5 shows its cost in Jacobi iteration. Tests with three different granularities are shown. Normally, a Filaments program uses the most natural granularity, which in this application is a filament per point. Because, the work per filament is very small (only a handful of instructions), using a filament per point is very expensive when execution of filaments is not optimized. In particular, the filament per point program on one node is 42% slower than a sequential program. The overhead decreases as the work per filament increases, but the potential parallelism, and consequently the flexibility, decreases. However, with run-time coarsening, the Filaments program with a filament per point is only 3% slower than the

Nodes	1	2	4	8
Non-overlapping	197	101	55.8	33.5
Overlapping	197	101	54.1	30.2

Figure 7: Overlapping communication and computation in Filaments. (Times in seconds.)

Nodes	1	2	4	8
Adapt	197	109	59.0	34.7
Filaments `BLOCK`	197	108	56.5	32.1

Figure 8: Implicit placement of data onto nodes. (Times in seconds.)

sequential program.

Figure 6 shows the cost of implicit communication. In this test, the Filaments program creates one filament per node, so there is not any overhead due to fine-grain execution. Thus, the two programs are essentially the same: each node updates a strip of the matrix in a 2-dimensional `for` loop. The primary difference between the two programs is that the Filaments program uses a DSM for (implicit) communication, which means additional overhead relative to explicit communication. For each row that is shared between nodes, the static program sends (or receives) a message. However, with implicit communication the program incurs a page fault, sends a request message, and sends the appropriate page of data. Furthermore, with the write-invalidate protocol, it is necessary to send invalidation messages when shared data is updated. Therefore, for each message in the coarse-grain program, there is a page fault and four messages (page request, page reply, invalidate, and invalidate acknowledgement) in the Filaments program. This overhead is obviously dependent on the application; in Jacobi iteration with a reasonably large matrix, the difference is not too large—most nodes send two messages on each iteration, a small cost compared to the massive amount of computation. (The eight-node test does not show as much overhead as the two- and four-node tests; this is an anomaly we cannot explain.)

Filaments' multithreading capabilities allow programs to overlap communication and computation. Figure 7 shows the benefit of multithreading in Jacobi iteration. The non-overlapping Filaments program uses a single pool, which means it is single-threaded. The overlapping Filaments program uses 3 pools: one for the top row, one for the bottom row, and the third for all other filaments. (All filaments in each of the first two pools reference the same remote page.) This achieves the maximal overlap, mitigating all wire and response time due to page faults, because there is sufficient work in the interior of the matrix. Overlapping results in a 10% improvement on eight nodes.

Figure 8 shows the overhead of Adapt in Jacobi iteration. The best data placement for Jacobi iteration is *block*, meaning that each node works on contiguous sets of n/p rows of the matrices. The Filaments program (without the Adapt subsystem) uses this placement, whereas the Adapt version determines a (variable) block placement after the first iteration[4]. Both programs use one pool of filaments per row, because this is how Adapt currently obtains the execution time of each row. As can be seen in the figure, the Adapt program performs slightly worse, due both to its own overhead and the slightly inexact block placement. However, as will be seen in Section 5.2, for adaptive problems an Adapt program can outperform a program with a static data placement.

[4]In some cases the bin-packing algorithm does not quite map n/p rows to each node, because small variances in row execution times lead to some nodes working on one more or one fewer row.

Nodes	1	2	4	8
Write-Invalidate	197	101	54.1	30.2
Writer-Owns	196	101	54.7	31.1

Figure 9: Implicit placement of data onto pages. (Times in seconds.)

Nodes	1	2	4	8
Static	191	98.1	51.7	29.0
Filaments Adapt	197	109	59.0	34.7
Filaments WO	196	101	54.7	34.3

Figure 10: Implicit decisions versus static decisions. (Times in seconds.)

Figure 9 compares the write-invalidate (WI) and writer-owns (WO) protocols. The WI program statically pads the data at the boundaries to ensure that there is no false sharing (and consequently, no thrashing). The data in WO program are contiguous in memory. Because each row fits exactly on one page, the WO program never needs to remap, explaining the small overhead compared to WI.

Figure 10 shows the sum total of making all decisions at run time versus making all decisions statically[5]. Both Filaments Adapt programs use run-time coarsening, implicit communication, and implicit overlap. Adaptive data placement and the writer-owns protocol are used exclusively by the respective programs. The static program uses one process per node, explicit message passing, explicit overlap, and explicit data placement (block). Even with the multiple overheads present in the Filaments Adapt and WO programs, both are still within 20% of the static program.

5.2 Additional Adapt Experiments

This subsection describes two additional experiments using Adapt. In the first, LU decomposition, a good data placement can be determined statically. The second, particle simulation, is an example of an application for which the best data placement changes during the computation.

LU decomposition is used to solve the linear system $Ax = b$. It is an example of application in which the load is not balanced. After a row is pivoted, it is never accessed again; on iteration i, only an $(n-i+1)$ by $(n-i+1)$ submatrix is accessed. On each iteration, the workload decreases by one row and every node must read the pivot row (row i), which is written by its owner. Communication is constant over all data placements. For these reasons, the best data placement for this application is `CYCLIC`.

The execution times for three versions of LU decomposition are shown in Figure 11. The Adapt program initially packs the bins in the variable block manner, just as in Jacobi iteration. However, Adapt quickly detects imbalanced load, re-enabling the fine-grain monitoring. At this point Adapt also detects a decreasing workload and packs the bins in a cyclic manner. The difference between this program and the Filaments program that uses `CYCLIC` is primarily the cost of the extra page faults necessary to change the data placement at run time. Because Adapt always starts with a block placement (as the default), this application acts as a worst-case. The Filaments block program is shown to indicate the load imbalance.

A Filaments program using Adapt can outperform a Filaments program using a static data placement for a whole class of adaptive applications. For example, Figure 12 shows the results

[5]The Adapt and WO subsystems have not yet been integrated, so we report on each separately.

Number of Nodes	1	2	4
Adapt	173	107	77.2
Filaments `CYCLIC`	172	95.0	68.2
Filaments `BLOCK`	172	111	84

Figure 11: LU decomposition, 512×512 (Times in seconds).

Number of Nodes	1	2	4	8
Adapt	69.4	40.1	29.8	23.5
Filaments 1 block per node	69.1	47.5	38.4	32.4
Filaments 2 blocks per node	69.1	47.0	39.1	25.3
Filaments 4 blocks per node	69.1	48.5	34.2	26.2
Filaments 8 blocks per node	69.1	46.5	39.3	42.6

Figure 12: Particle Simulation, grid 64×64, 150 particles. (Times in seconds.)

from a particle simulation. The basic structure of this application mimics the behavior of MP3D [McD88]. Several molecules move through a two-dimensional grid of cells, colliding with other molecules that occupy the same cell. Colliding molecules move to a random location on the grid, which can in practice lead to a clustering of particles in certain regions of the grid. Our experiment is configured so that this is exactly what happens.

The Adapt program uses a variable block placement and periodically remaps the grid to balance the number of particles (for this particular program Adapt performed three remappings). We tested several Filaments programs with different static data placements; using larger block sizes exacerbates the load imbalance and using smaller block sizes causes excess communication. None of the Filaments programs with a static placement performs as well as the Adapt version. Particle simulation is thus an example of an application for which Adapt can outperform any static data placement.

5.3 Additional WO Experiments

This section describes two additional experiments showing the benefits of the writer-owns protocol. The first experiment, matrix multiplication, shows that WO performs well even in non-iterative applications. The second, Jacobi iteration, illustrates the advantage of eliminating instead of tolerating false sharing in an iterative computation. In both of these tests, the row size is 500, which occupies slightly less than one page. This row size induces false sharing if the data are placed contiguously in memory. Therefore, the WI programs have to (statically) pad the data at the boundaries to ensure that there is no false sharing.

Matrix multiplication computes $C = A \times B$, where A, B, and C are $n \times n$ matrices. Each node computes a horizontal contiguous strip of rows of the C matrix. A master node initializes the matrices, and the other nodes fault on all of B and the appropriate parts of A. The execution times for matrix multiplication are shown in Figure 13. This application is not iterative, so WO suffers from not being able to amortize its overhead of relocating data; however, the penalty for using WO is quite small.

The second test is Jacobi iteration. Because Jacobi iteration uses two grids and writes to one grid every other iterations, the WO program eliminates all false sharing in the first two iterations. In contrast, the WS program has to merge diff lists on each iteration, explaining the additional

Nodes	1	2	4	8
Write-Invalidate	224	119	64.7	44.3
Writer-Owns	228	121	66.3	45.3

Figure 13: Matrix Multiplication, 500 $\times$ 500 (Times in seconds.)

Nodes	1	2	4	8
Write-Invalidate	165	94.1	49.6	32.1
Writer-Owns	165	94.1	48.9	31.9
Write-Shared	166	99.8	52.2	44.0

Figure 14: Jacobi iteration, 500 $\times$ 500, 100 iterations (Times in seconds.)

overhead relative to the WO program. Figure 14 shows that WS performs worse than WO, and that its relative performance decreases as the number of node increases. The write-invalidate protocol serves as a baseline, because it pads pages and hence avoids thrashing.[6] However, as discussed above, static padding is nontrivial.

6 Summary and Conclusions

This paper has examined static and dynamic approaches to implementing parallel applications on distributed-memory machines. Specifically, we have looked at four issues that can be addressed statically when a program is developed or compiled or that can be addressed dynamically at run time:

- the granularity of parallelism (coarse or fine),
- how data is exchanged between processors (directly using explicit message passing or indirectly using shared variables),
- overlapping communication and computation (explicitly using message passing or implicitly using a distributed shared memory), and
- placing data on nodes and on pages (explicitly in the source code or implicitly during execution).

Section 2 showed that using run-time decisions leads to a simpler programming interface, mainly because decisions are implicit.

Section 3 described how concurrency and communication can be addressed at run time. In particular, we discussed how to coarsen the granularity of parallelism, implement a distributed shared memory using multiple server threads, and use multithreading to overlap communication and computation.

Section 4 described how data placement issues can be addressed at run time. The Adapt system monitors communication overhead, computation time, and page reference patterns to decide how to place pages on nodes. The writer-owns protocol decides during execution how to place data on

[6]The WO program is faster than the WI program, which we believe is due to the different memory layout and hence different caching effects. Because WO operates the same as WI after it has eliminated false sharing, we would expect WI with padding to be faster than WO in most applications.

pages in order to avoid false sharing. Both make it possible to adapt to the run-time characteristics of an application, and hence they make it possible to make better decisions than could possibly be made with the lesser amount of information that is available at compile time.

Section 5 presented performance figures, both to show the cost of individual mechanisms and to show their combined effect. Dynamic mechanisms inherently result in overhead, because they monitor events at run time. However, each mechanism we have discussed either adds only a minor amount of overhead to the overall execution time or in some cases improves the overall time. The composite effect of using all mechanisms for the same application is, of course, additive, but they do not have to be used together. In particular, the individual mechanisms could be used in other systems that employ fine-grain concurrency and/or distributed shared memory.

In conclusion, it is indeed possible to make effective decisions at run time. The costs are a small amount of execution overhead and a somewhat larger run-time system. The benefits are a much simpler interface and potentially better performance.

References

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

[BKT90] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with Distributed Programming in Orca. In *Proc. of the 1990 Int'l Conf. on Computer Languages*, pages 79–89, March 1990.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *COMPCON '93*, pages 528–537, 1993.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium On Operating Systems*, pages 152–164, October 1991.

[DJAR91] Partha Dasgupta, Richard J. LeBlanc Jr., Mustaque Ahmad, and Umakishore Ramachandran. The Clouds distributed operating system. *Computer*, 24(11):34–44, November 1991.

[EAL93] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Shared filaments: Efficient support for fine-grain parallelism on shared-memory multiprocessors. Technical Report 93-13, Dept. of Computer Science, University of Arizona, April 1993.

[EZ93] Derek L. Eager and John Zahorjan. Chores: Enhanced run-time support for shared memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[FLA94] 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.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: a coherent distributed shared memory design. In *Proceedings of 12th ACM Symposium On Operating Systems*, pages 211–223, December 1989.

[Fre95] Vincent W. Freeh. Writer-Owns: a new page consistency protocol for dynamically controlling thrashing on distributed-shared memory systems. December 1995.

[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.

[Har64] Francis H. Harlow. The particle-in-cell computing method for fluid dynamics. In Bernie Alder, editor, *Methods in Computational Physics*, pages 319–343. Academic Press, Inc., 1964.

[HB92] Matthew Haines and Wim Böhm. The design of VISA: A virtual shared addressing system. Technical Report CS-92-120, Colorado State University, May 1992.

[HFM88] D. Hansgen, R. Finkel, and U. Manber. Two algorithms for barier synchronization. *Int. Journal of Parallel Programming*, 17(1):1–18, February 1988.

[HKT92] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[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.

[HPF93] High Performance Fortran language specification. October 1993.

[KDCZ94] 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.

[KK94] Ken Kennedy and Ulrich Kremer. Automatic data layout for High Performance Fortran. Technical Report CRPC-TR94498-S, Rice University, December 1994.

[LA95] David K. Lowenthal and Gregory R. Andrews. Adaptive data placement for distributed-memory machines. TR 95-13, University of Arizona, November 1995.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[McD88] Jeffrey D. McDonald. A direct particle simulation method for hypersonic rarified flow. Technical Report 411, Stanford University, March 1988.

[Mos93] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, January 1993.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1994.

[TC88] Robert H. Thomas and Will Crowther. The Uniform system: an approach to runtime support for large scale shared memory parallel processors. In *1988 Conference on Parallel Processing*, pages 245–254, August 1988.

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