Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations *

Vincent W. Freeh, David K. Lowenthal, Gregory R. Andrews
{vin,dkl,greg}@cs.arizona.edu
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

A fine-grain parallel program is one in which processes are typically small, ranging from a few to a few hundred instructions. Fine-grain parallelism arises naturally in many situations, such as iterative grid computations, recursive fork/join programs, the bodies of parallel `FOR` loops, and the implicit parallelism in functional or dataflow languages. It is useful both to describe massively parallel computations and as a target for code generation by compilers. However, fine-grain parallelism has long been thought to be inefficient due to the overheads of process creation, context switching, and synchronization. This paper describes a software kernel, Distributed Filaments (DF), that implements fine-grain parallelism both portably and efficiently on a workstation cluster. DF runs on existing, off-the-shelf hardware and software. It has a simple interface, so it is easy to use. DF achieves efficiency by using stateless threads on each node, overlapping communication and computation, employing a new reliable datagram communication protocol, and automatically balancing the work generated by fork/join computations.

1 Introduction

One way to write a program for a parallel machine is to create a process (thread) for each independent unit of work. Although generally thought to be inefficient, such *fine-grain* programs have several advantages. First, they are architecture independent in the sense that the parallelism is expressed in terms of the application and problem size, not in terms of the number of processors that might actually be used to execute the program. This also makes fine-grain programs easier to write, because it is not necessary to cluster independent units of work into a fixed set of larger tasks; indeed, adaptive programs such as divide-and-conquer algorithms do not have an *a priori* fixed set of tasks. Third, the implicit parallelism in functional or dataflow languages is inherently fine-grain, as is the inner-loop parallelism extracted by parallelizing compilers or expressed in parallel variants of languages such as Fortran; using fine-grain threads simplifies code generation for such languages. Finally, when there are many more processes than processors, it is often easier to balance the total amount of work done by each processor; in a coarse-grain program, it is important that each processor be statically assigned about the same amount of work, but this is impossible if the computation is dynamic or if the amount of work per process varies.

We have developed a software kernel called Filaments that strives to support efficient execution of fine-grain parallelism and a shared-memory programming model on a range of multiprocessors. A *filament* is a very lightweight thread. Each filament can be quite small, as in the computation of an average in Jacobi iteration; medium size, as in the computation of an inner product in matrix multiplication; or large, as in a coarse-grain program with a process per processor. The Filaments package provides a minimal set of primitives that we have found sufficient to implement all the parallel computations we have examined so far. As an analogy, the goal of Filaments relative to other approaches to writing parallel programs is similar to the goal of RISC relative to other styles of processor architecture: to provide a least common denominator that is easy to use as a compiler target and that is efficiently implementable.

*This research was supported by NSF grants CCR-9108412 and CDA-8822652

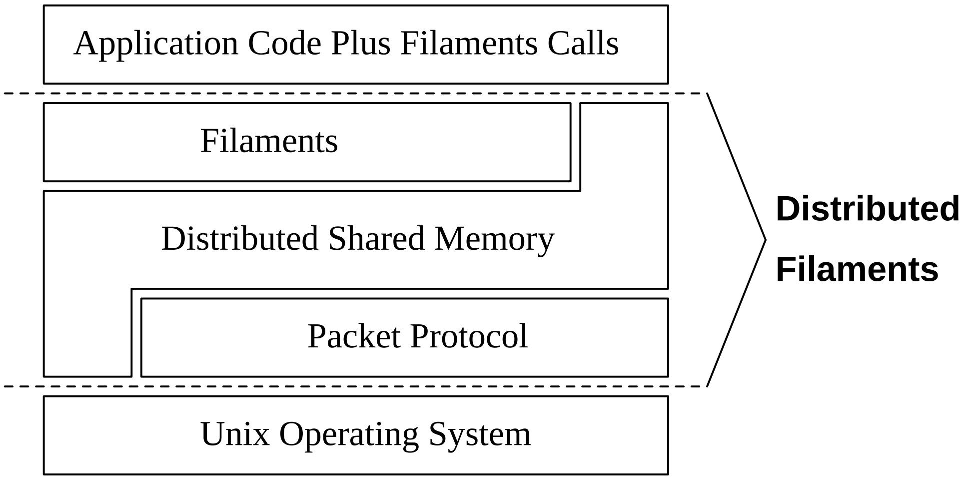


Figure 1: Distributed Filaments Components

Previous work has described the Shared Filaments (SF) package for shared-memory multiprocessors [EAL93]. We have used SF as a system-call library for a variety of applications; performance using SF is typically within 10% of that of equivalent coarse-grain programs, and it is sometimes even better (for load-imbalanced problems). We have also used SF as the back end for a modified Sisal compiler, thereby achieving efficient *forall* and function-call parallelism in a dataflow language [Fre94].

This paper addresses the issue of providing portable, efficient fine-grain parallelism on a cluster of workstations. Distributed Filaments (DF) extends SF and combines it with a distributed shared memory (DSM) customized for use with fine-grain threads. Figure 1 shows the components of DF and their inter-relation. The unique aspects of DF are:

- a multi-threaded DSM that implicitly overlaps communication and computation by executing new filaments while page faults are being serviced;
- a new page consistency protocol, implicit invalidate, for regular problems;
- a low overhead, reliable, datagram communication package, and
- an efficient implementation of the fork/join programming paradigm.

DF requires no special hardware support for a shared-memory address space or for multithreading. In addition, the only machine-dependent code in DF is a very small amount of context-switching code; hence, DF can easily be ported to different distributed-memory architectures (see below).

DF uses multithreading to overlap communication and computation, thereby masking communication latency due to "wire" and memory-access (DMA) time. In particular, when a thread faults on a page, another thread can execute while the page request is still outstanding. In this way DF differs from DSMs such as IVY [LH89], Munin [CBZ91], Mirage [FP89], and Midway [BZS93], where a page fault blocks the entire node. (VISA [HB92], a DSM specifically supporting distributed SISAL programs, does some overlapping of computation and communication, but VISA's stack-based nature limits the overlap relative to the multithreading of DF.) Overlapping communication and computation is useful on both older (e.g. Ethernet) and newer (e.g. FDDI, ATM) network technologies. While FDDI and ATM provide higher bandwidth than Ethernet, there is still sufficient latency to make overlapping beneficial [TL93].

DF also uses a reliable datagram protocol built on UDP to reduce communication overhead. Our protocol buffers only short request messages, saving both time and space. This protocol provides reliable communication with the efficiency and scalability of UDP.

Another unique aspect of DF is its fork/join mechanism, which makes DF much easier to use for recursive applications. For adaptive quadrature and evaluating binary expression trees, for example, the simplest way to implement parallelism is to execute recursive procedure calls in parallel. However, it can be difficult to implement fork/join efficiently, because recursive parallelism can result in many small tasks, numerous messages to communicate arguments and results, and differing task workloads. DF solves each of these problems.

Distributed Filaments runs on a cluster of Sun workstations on top of SunOS; prototypes run on a cluster of DECstations running Mach and on an Intel Paragon running OSF. Performance on the Suns is such that an eight node version of Jacobi iteration—which has a large number of very small threads, and thus is a worst-case application—is within 8% of an equivalent coarse-grain program that uses explicit message-passing. DF achieves a speedup of 5.58 on 8 nodes relative to the sequential program. Moreover, overlapping communication and computation results in a 21% improvement over the equivalent non-overlapping program.

The rest of the paper is organized as follows. The next section gives an overview of the Filaments package. Section 3 describes our multi-threaded DSM, the datagram communication package, and the implicit-invalidate page consistency protocol. Section 4 summarizes performance results. Section 5 discusses related work. Finally, Section 6 gives concluding remarks.

2 Filaments Overview

The Filaments package supports fine-grain parallelism on both shared- and distributed-memory machines. The same user code (C plus Filaments calls) will run unchanged on either type of machine. However, the run-time systems that support the shared and distributed versions differ. Below we first describe elements common to both implementations and then describe the important elements of the distributed version. (We will refer to the shared and distributed implementations of Filaments as SF and DF, respectively.)

2.1 Common Elements

There are three kinds of filaments: run-to-completion, iterative, and fork/join. These are sufficient to support all the parallel applications we have examined, and hence we believe they are sufficient for most if not all applications.

A run-to-completion (RTC) filament executes once and then terminates; it is used in applications such as matrix multiplication. Iterative filaments execute repeatedly, with barrier synchronization occurring after each execution of all the filaments; they are used in applications such as Jacobi iteration. Fork/join filaments recursively fork new filaments and wait for them to complete; they are used in divide-and-conquer applications such as adaptive quadrature.

A filament does not have a private stack; it consists only of a code pointer and arguments. All filaments are independent; there is no guaranteed order of execution among them. Filaments are executed one at a time by server threads, which are traditional threads with stacks. Our threads package is based on the one used in the SR run-time system [AOC+88]. It is non-preemptive, and it employs a scheduler written specifically for DF.

Many Filaments programs attain good performance with little or no optimization. However, achieving good performance for applications that possess many small filaments requires three techniques: inlining, pruning, and pattern recognition. These techniques reduce the cost of creating and running filaments, reduce the working set size to make more efficient use of the cache, and produce code that is amenable to subsequent compiler optimizations of the Filaments code.

Inlining and pruning eliminate some of the overhead associated with creating and running filaments. Static inlining is performed for RTC and iterative filaments. Instead of calling the function specified by the filament each time, the body is inlined in a loop over all filaments. Dynamic pruning is performed for fork/join filaments. When enough work has been created to keep all nodes busy, forks are turned into procedure calls and joins into returns.

Inlining eliminates a function call, but DF still has to traverse the list of filament descriptors to obtain the arguments. The Filaments package can produce code that is amenable to compiler optimizations, improves cache performance, and eliminates additional overhead associated with running filaments. It does this by recognizing common patterns of RTC and iterative filaments at run-time. Currently, Filaments recognizes contiguous strips of one- or two-dimensional arrays of filaments assigned to the same node. For such strips of filaments the package dynamically switches to code that iterates over the assigned strip, generating the arguments in registers rather than reading the filament descriptors. This optimization supports a large subset of regular problems; we are working on supporting other common patterns.

2.2 Run-to-Completion and Iterative Filaments in DF

The possibility of a DSM page fault means filaments in DF must be able to block at unpredictable points. Hence, DF creates multiple server threads per node (as opposed to SF, which creates one server thread per node). A server thread runs filaments until either a page fault occurs or all eligible filaments have been executed. When a fault occurs, the state of the filament is saved on the stack of its server thread, and another server thread is executed. The faulting server thread is inserted in a queue for the appropriate page. On receipt of a page, all server threads waiting on that page are enabled.

Two important issues arise with RTC filaments in DF: avoiding excessive faulting and overlapping communication and computation as much as possible. DF provides *pools* to address both issues.

A pool is a collection of filaments that ideally reference the same set of pages. At creation time, the

Filaments program (user or compiler generated) assigns a filament to a pool. When a program is started, a server thread on each node starts executing a pool of filaments. On a page fault, 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.

Iterative filaments are a generalization of RTC filaments. For iterative applications, DF ensures that after the first iteration the pools that are run first will be those that faulted on the previous iteration; i.e., the faults are frontloaded, which increases the potential for overlap of communication and computation. This is a useful optimization, as many iterative applications have constant sharing patterns. DF implements this 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. 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. The combination of these two mechanisms effectively frontloads the faults, and the greatest overlap of communication and computation will be achieved.[1]

At present, it is up to the programmer or compiler to determine the number of pools on each node and to assign filaments both to a node and to a pool on that node. Both of these decisions can be nontrivial and have to be done well or performance will suffer (although correctness will not). The filaments should be assigned to nodes so that the load is balanced, and the filaments within a node should be assigned to pools so that faults are minimized and good overlap of computation and communication is achieved. Determining the correct node and pool for filaments can lead to some of the same difficulties that occur in writing a coarse-grain program. We are currently working on adaptive algorithms for making both of these decisions within DF at run time.

2.3 Fork/Join Filaments in DF

In recursive, fork/join applications, the computation starts on just one node, and other nodes are idle.

[1] 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 frontload the faults by running one filament from each pool at the beginning of each iteration.

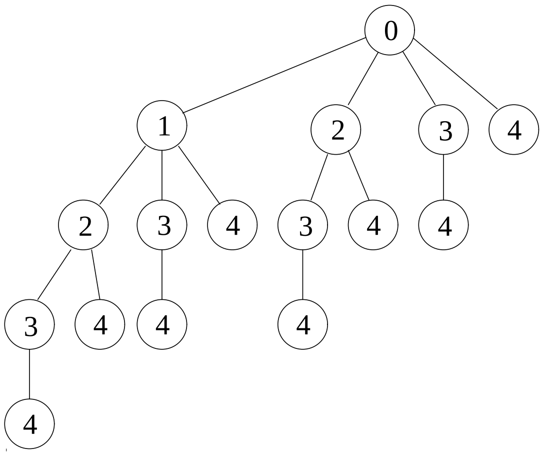


Figure 2: Logical tree of 16 nodes, for applications that create parallelism by the pattern of 2 forks and a join. At each step, the number of nodes with work doubles. The numbers in the figure indicates the step in which the node first gets a filament.

To get all nodes involved in the computation, new work (from forks) needs to be sent to idle nodes. Dynamic load balancing may then be required to keep the nodes busy, because different nodes can receive different amounts of work.

Load balancing can, however, have several negative effects. In DF, there are two significant ones. First, when a node starts executing a filament, it needs to get the data associated with the filament. At best this costs the time to acquire the working set of the filament, and at worst it can lead to thrashing. Second, the initial load-balancing phase can be very costly, as all nodes other than the one that starts the computation will either query other nodes with requests for work (most of which will be denied) or flood the initial node with requests (which can cause a bottleneck).

DF first uses a sender-initiated load-balancing scheme and then employs a simple receiver-initiated, dynamic load-balancing policy (which can be enabled and disabled by the application programmer or compiler). Suppose that a fork/join application creates parallelism by two forks and then a join. The initial load-balancing phase then works as follows. The nodes form a logical tree (see Figure 2). The root begins the computation; after forking the first two filaments, it sends one filament to its left child and keeps the other. Both nodes continue the computation. When the root forks another two filaments, it now sends one to its second child and keeps the other; the next time it sends a new filament to its third child and keeps the other; and so on. Each child node follows the same pattern. Consequently, in each step the number of nodes with work doubles. The initial phase continues in this way until a node has sent work to all of its children, after which it keeps all filaments it forks.

After the initial work-distribution phase, some applications will need to employ dynamic load balancing; in DF this is receiver initiated. When a node has no new filaments and none that are suspended waiting for a page, it queries other nodes in a round-robin fashion. For applications that do not exhibit much load imbalance—such as evaluating balanced binary expression trees, merge sort, or recursive FFT—the cost of acquiring pages outweighs the gain of load balancing. On the other hand, for applications such as adaptive quadrature—where evaluating intervals can take widely varying amounts of time—dynamic load balancing is absolutely necessary.

Another concern in implementing fork/join is avoiding thrashing, which can occur when two nodes write to the same page. Fork/join in DF uses two mechanisms. The first is similar to one used in the Mirage [FP89] system. A node will keep a page for a certain length of time before giving it up; during that time all requests for the page from other nodes are dropped (they will be retransmitted later). The Mirage timer can hurt system performance because it may delay sending a requested page. However, this problem is mitigated in Filaments by multithreading; if there is other work, the delay of the request is insignificant. The second mechanism used to control thrashing is that when a page arrives, all server threads waiting on the page are scheduled at the *front* of the ready queue. Hence the page that just arrived will be utilized as soon as the currently executing thread gives up control. This increases the probability that the page is still resident by the time the enabled threads are actually scheduled to run.

3 Distributed Shared Memory

Our multi-threaded distributed shared memory (DSM) is built on top of SunOS and therefore requires no specialized hardware or changes to the operating system kernel. In a single-threaded DSM implementation, all work on a faulting node is suspended until the fault is satisfied. In a multi-threaded implementation, other work is done while the remote fault is pending. This makes it possible to overlap communication and computation.

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, while 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.

The shared address space is divided into individually protected pages of 4K bytes each (this is the granularity supported in SunOS). However, a user does not have to use all of the locations in a page. In particular, user data structures can be padded to distribute elements onto different pages. We have written a library routine that allocates a data structure in global memory and automatically pads (when necessary). Additionally, two or more pages can be grouped so that a request for any page in the group is a request for all of them. Thus our DSM supports pages that can have different sizes than the pages directly supported by the operating system.

There are two events 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 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 `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.

There are any number of page consistency protocols (PCP) that could be implemented. We have found three PCPs to be sufficient to support the wide range of applications we have programmed in DF: *migratory* [CBZ91], *write-invalidate* [LH89], and a new protocol we call *implicit-invalidate*. The migratory PCP keeps only one copy of each page; the page moves from node to node as needed. Write-invalidate allows replicated, read-only copies; all are invalidated explicitly when any copy is written. Implicit-invalidate is similar to write-invalidate, but it is optimized to eliminate the invalidation messages. In particular, read-only copies of a page are *implicitly* invalidated at every synchronization point. Hence a read-only copy of a page has a very short lifetime and explicit invalidate messages are not needed. Implicit-invalidate works only for regular problems with a stable sharing pattern, such as Jacobi iteration.

A DSM requires reliable communication, ideally with low overhead. On a Unix system there are two communication choices: TCP and UDP. TCP is reliable, but it does not scale well with the number of

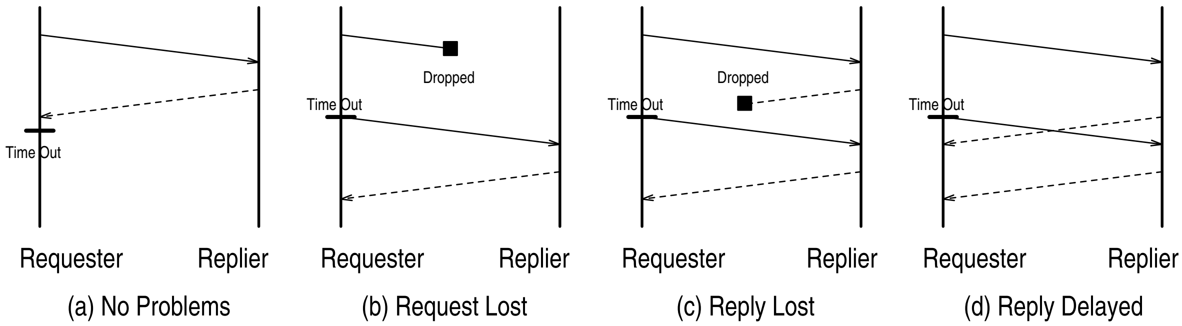


Figure 3: Possible scenarios in Packet

nodes; UDP is scalable, but it is not reliable. UDP is also slightly faster than TCP, and it supports message broadcast. Therefore we use UDP and an efficient reliability protocol to create Packet, a low overhead reliable datagram communication package.

In Packet there are two types of messages: *request* and *reply*. Communication always occurs in pairs: first a request and then the associated reply. When either the request or reply message is lost or delayed, the request message is retransmitted. All request messages are buffered; however, they are very short (20 bytes or less), so buffering consumes very little time and space. Packet is efficient because only small messages are buffered and, in the common case, only two messages are sent. (Figure 3 shows the four possible situations that can arise.) In an unreliable network where a lost packet is a likely event, a different reliability mechanism—such as the one in TCP—might perform better than Packet. However, in a highly reliable network, Packet adds essentially no additional overhead. In particular, the overhead consists of buffering request messages and, when a reply is received, removing the message from a list. The list is never longer than the number of messages that are sent between synchronization points.

With Packet we avoid the expense of buffering DSM page data. All page replies are constructed using the current contents of the page. Nodes delay at synchronization points until all outstanding page requests have been satisfied. Therefore, in a program without race conditions, a page request always returns a consistent copy of a page.

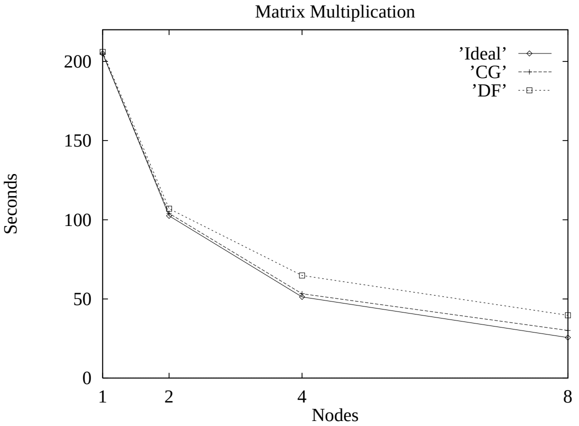
Packet is similar to the VMTP protocol [CZ83], which also does not buffer reply messages and retransmits request messages only when necessary. However, there is a fundamental difference: VMTP uses a synchronous send/receive/reply, whereas Packet uses an asynchronous send/receive/reply.

Because request messages are retransmitted until acknowledged, we can implement a very efficient mutual exclusion mechanism. When a server thread is in a critical section, all messages that cause critical data (i.e., thread queues) to be modified are ignored—they will be retransmitted when the requester times out. (The servicing of some messages, such as a page request, will never modify critical data.) The entry and exit protocols for a critical section are simply a single assignment statement, so many, very small critical sections can be used efficiently. Our experience shows that an ignored message is so rare that it essentially never occurs.

A *reduction* in Filaments is a primitive that both (1) causes a value from each node to be accumulated and then disseminated to all nodes, and (2) ensures that no node continues until all nodes have completed the same reduction. Hence, a reduction also serves as a barrier synchronization point. (A "pure" barrier is a reduction that does not compute a value.) A reduction is a high-level mechanism that we have used in our application programs instead of low-level mechanisms like locks. Reductions are not necessarily a part of a DSM, although they are a necessary part of a parallel programming system. Implementing reductions as part of the DSM has the advantage that they can be an integral part of the page consistency protocol (PCP). This can reduce the number of messages required to maintain consistency and allows greater flexibility in the design of a PCP. For example, in our implicit-invalidate protocol, a node invalidates all its read-only copies before performing an inter-node barrier; hence, no invalidation messages are sent.

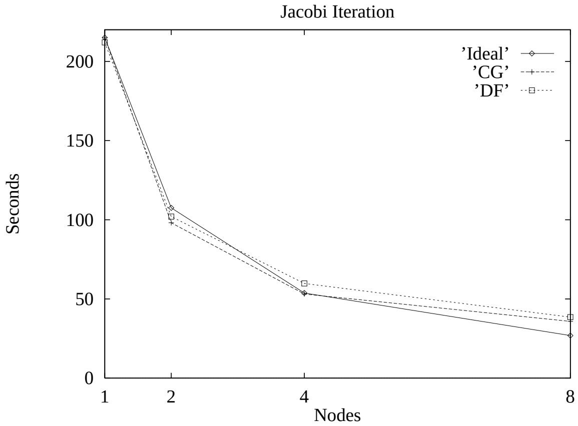
4 Performance

This section reports the performance of four programs: matrix multiplication, Jacobi iteration, adaptive quadrature, and evaluation of binary expression trees. For each we developed a sequential program, a coarse-grain program, and a Filaments program. All programs use similar computation code. The sequential programs are distinct from the others—they are not simply a parallel program run on one node. The coarse-grain (CG) programs have a single heavy-weight process on each node and use explicit message-



Nodes	1	2	4	8
CG Time (sec)	205	104	53.3	30.1
CG Speedup	1.00	1.97	3.85	6.81
DF Time (sec)	206	107	64.8	39.7
DF Speedup	1.00	1.92	3.16	5.16

Figure 4: Matrix multiplication, size 512×512. Sequential program: 205 sec.



Nodes	1	2	4	8
CG Time (sec)	215	98.1	53.1	35.8
CG Speedup	1.00	2.19	4.05	6.01
DF Time (sec)	212	102	59.8	38.5
DF Speedup	1.01	2.11	3.60	5.58

Figure 5: Jacobi iteration, 256×256, $\epsilon = 10^{-3}$, 360 iterations. Sequential program: 215 sec.

passing. They use UDP for communication. The times reported are those of the tests in which all messages were delivered and the program completed; when a message was lost, the program hung and the test was aborted. All speedups are computed using the sequential program as the baseline. In the graphs that follow, the ideal time is the sequential time divided by the number of nodes.

Below, we briefly describe four applications, present the results of runs on 1, 2, 4, and 8 nodes, and examine the parallel speedup. In Section 4.5 we examine the overheads of Filaments in detail. All tests were run on a network of 8 Sun IPCs connected by a 10Mbs Ethernet. We used 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`. The variance of the test runs was small. The tests were performed when the only other active processes were Unix daemons.

4.1 Matrix Multiplication

The execution times for matrix multiplication are shown in Figure 4. The programs compute $C = A \times B$, 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 distributes all of B and the appropriate parts of A to the other nodes.

In the coarse-grain program, all slave nodes receive all the data they need before starting their computation. The distribution of the A and B matrices takes 5.1 seconds in the 8 node program. This initial overhead limits the speedup of the coarse-grain program.

The DF program for matrix multiplication uses run-to-completion filaments, one per point of the C matrix, and the write-invalidate PCP. Because there are no write conflicts in the C matrix, there is very little synchronization overhead (two barriers, one to ensure the master node initializes A and B before other nodes start computing, and one to ensure all nodes have computed their part of C before the master node prints it). However, in the problem tested each matrix requires 512 pages of 4K bytes each. All $p - 1$ slave nodes must receive all of B and $1/p$ of A, so the number of page requests is $O(pn^2)$ in the DF program. The master node must service all these page requests. The overhead to service 4032 page requests in the 8 node program is approximately 6.2 seconds, and the network is saturated by the large number of messages. This saturation results in an increase in the latency of acquiring pages and leads to workload imbalance. These factors explain the drop-off in speedup on 4 and 8 nodes for the DF program.

4.2 Jacobi Iteration

Laplace's equation in two dimensions is the partial differential equation $\nabla^2(\Phi) = 0$. Given boundary values for a region, its solution is the steady-state values of interior points. These values can be approximated numerically by using a finite difference method such as Jacobi iteration, which repeatedly computes new values for each point, then tests for convergence.

In Jacobi, one horizontal strip of each array is distributed to each node. The nodes in the coarse-grain program repeatedly send edges, update interior points, receive edges, update edges, and check for termination. In this way the coarse-grain program achieves maximal overlap of communication and computation.

The DF program uses iterative filaments—one per point—and three pools of filaments—one each for the top row, bottom row, and interior rows. Filaments in the top and bottom pools fault, generating a page request; those in the local pool do not fault. All communication latency will be overlapped provided that the latency of acquiring pages is less than the total execution time of filaments in the local pool; the DF program achieves full overlap. A reduction is needed on every iteration.

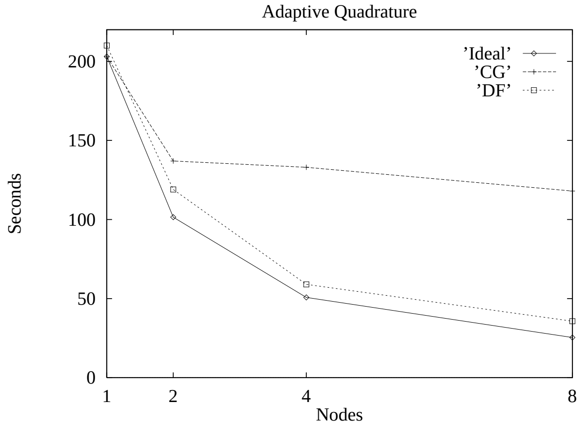
Figure 5 shows the main results of our Jacobi iteration programs.[2] The coarse-grain program gets better than linear speedup for 2 and 4 nodes (the primary reason is the size of the working set and its effect on the cache, etc. [SHG93]), and it gets reasonable speedup on 8 nodes. The gain of overlapping communication and computation in the coarse-grain program is 5.5% and 14% on 4 and 8 nodes, respectively.

The DF program uses the implicit-invalidate PCP, which eliminates invalidation messages. The speedup obtained is 3.60 on 4 nodes and 5.58 on 8 nodes. The running times increase by 10% and 26% on 4 and 8 nodes, respectively, if the communication latency is not overlapped.

4.3 Adaptive Quadrature

Adaptive quadrature is an algorithm to compute the area under a curve defined by a continuous function. It works by dividing an interval in half, approximating the areas of both halves, and then subdividing further if the approximation is not good enough. The programs tested evaluate a curve that causes workload imbalance.

Our coarse-grain approximation divides the interval into p subintervals and assigns one to each node. However, this can lead to severe load imbalance, as reflected in Figure 6. A second program that uses a bag-of-tasks [CGL86] has better speedup, but its absolute time is much worse. The overhead of accessing the centralized bag is extremely high due to the large number of small tasks. These coarse-grain programs illustrate the need for a low-overhead, decentralized load balancing mechanism.



Nodes	1	2	4	8
CG Time (sec)	203	137	133	118
CG Speedup	1.00	1.48	1.53	1.72
DF Time (sec)	210	119	59.0	35.7
DF Speedup	0.97	1.71	3.44	5.69

Figure 6: Adaptive quadrature, interval of length 24. Sequential program: 203 sec.

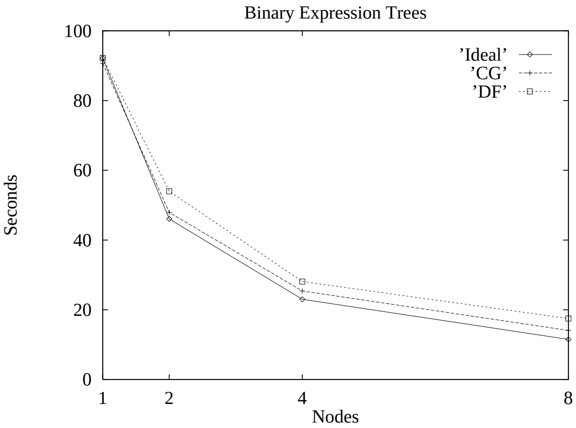
The natural algorithm for this problem uses divide-and-conquer, so our DF program uses fork/join filaments. Speedups of 3.44 and 5.69 were obtained on four nodes and eight nodes, respectively. The speedup tapers off as the number of nodes increases because the two nodes evaluating the extreme intervals initially contain most of the work. With eight nodes, six complete their initial work quickly and make load-balancing requests. This not only increases the number of messages but increases the likelihood of a load-balance denial (because only two have sufficient work).

4.4 Binary Expression Trees

The fork/join paradigm can also be used to compute the value of a binary expression tree, an application described in [EZ93]. The leaves are matrices and interior operators are matrix multiplication; the tree is traversed in parallel and the matrices are multiplied sequentially. Figure 7 contains the results of running the matrix expression program with 70 by 70 matrices and a balanced binary tree of height 7. The maximum possible speedup that can be achieved for this application is limited by the tail-end load imbalance. In particular, near the top of the tree some nodes must remain idle. However, because the work doubles with each level of the tree, good speedup can still be achieved. For the problem tested, the maximum speedup is 3.85 and 7.06 on 4 and 8 nodes, respectively.

The coarse-grain program contains two phases. In

[2] The DF program should not run faster than the sequential program, but it does. Even though the C code for the computation of each point is identical in all three programs, the compiler generates different assembler code for DF than for the CG and sequential programs. The innermost computation (the average) is slightly faster in DF than in the other two programs. This anomaly, for which we have no explanation, also occurs on some other test programs.



Nodes	1	2	4	8
CG Time (sec)	90.7	47.9	25.4	14.1
CG Speedup	0.99	1.92	3.63	6.53
DF Time (sec)	92.2	54.0	28.1	17.5
DF Speedup	1.00	1.71	3.28	5.26

Figure 7: Binary expression tree evaluation, 70×70 matrices, tree of height 7. Sequential program: 92.1 sec.

Nodes	2	4	8
Time (msec)	3.20	5.29	8.45

Figure 8: Barrier synchronization, 1000 barriers

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
Page faults	4,120	238.

Figure 9: Filaments overheads

the first, the program divides the work up evenly among the nodes. The second phase combines the intermediate values that each node calculated into a single result using a tree. Tail-end load imbalance is handled in this last phase. Proceeding towards the top of the combining tree, half of the active nodes become inactive at each level. There are very few messages in this algorithm (a total of $2(p-1)$ to transfer result matrices in the combining tree). The overhead is very low, so the speedup is very good.

This DF fork/join program uses the global (DSM) memory, unlike adaptive quadrature where all the information is contained in the function parameters. The migratory PCP is used, although in this particular application there is no performance difference between migratory and write-invalidate. The DF program sends many more messages than the coarse-grain program because (1) the parallelism begins from a single root filament and its children are distributed to the other nodes and (2) data is acquired implicitly by page faults (requiring a request and a reply).

4.5 Analysis of Overhead

DF introduces four categories of overheads relative to sequential programs: filaments execution, paging, synchronization, and workload imbalance. Filaments execution has the following costs: creating and running filaments, more memory for filament descriptors (and therefore less effective caching), and lost compiler optimizations. As discussed in Section 2.1, the Filaments package has support for decreasing these costs. This support resulted in Filaments execution overhead of less than 5% in all the test programs.

The second source of DF overhead is due to DSM paging. The faulting node incurs the cost of faulting on the page and sending and receiving the resulting messages. The owner of the requested page incurs the cost of servicing the page request (receiving the request and sending the page). The paging overhead per node is application-dependent. In general it does not depend on the number of nodes but on the sharing of data. However, even with a constant paging overhead per node, the total number of messages in the system will increase linearly with the number of nodes. This can be a problem when the network becomes saturated, as is the case in Jacobi iteration on eight nodes.

The third DF overhead is due to synchronization, which results from barriers (in RTC and iterative filaments) or messages containing filaments and result values (in fork/join filaments). The overhead of barriers is a function of the number of nodes. DF 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 8. 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. The barrier time given includes message latency (wire time) as well as message processing overhead. Communication latency cannot be overlapped with computation during a barrier because a processor has no more work to do when it reaches a barrier. Therefore, a barrier is an expensive operation.

The final DF overhead is a result of nodes having differing workloads. This can occur in two ways: either nodes are given different amounts of work, or the work leads to different amounts of paging and synchronization overhead. Differing workload results in nodes

Node	Master	Interior	Tail
Work	22.3	22.9-24.4	22.6
Filament Exec	1.57	1.54-1.87	1.73
Data Transfer	7.75	2.31-3.02	1.53
Sync Overhead	0.99	1.51-2.14	1.12
Sync Delay	6.62	5.24-10.3	14.7

Figure 10: Analysis of overheads in Jacobi iteration, 8 nodes, 256×256, $\epsilon = 10^{-3}$, 360 iterations. Total execution time 42.1 seconds.

arriving at synchronization points at different times and hence leads to uneven and longer delays.

The coarse-grain programs also incur overheads due to data transfer, synchronization, and workload imbalance. The latter two overheads are roughly the same as DF; however, the data-transfer overhead is much less because the coarse-grain programs use explicit messages to transfer data.

Some of the filaments overheads are shown in Figure 9. Each overhead 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 is more than five times faster than not inlining. For comparison purposes, context switch times for the light-weight server threads are shown as well. The page fault times assume the owner is known and the page is immediately available.

Figure 10 shows DF overheads in the specific case of Jacobi iteration. The nodes are split into three categories: the master node, the interior nodes, and the tail node. The master node and the tail node fault on one page and service one page request per iteration. Interior nodes incur two page faults and two page requests per iteration. In addition, the master node must service all the initial page requests. The execution time in Figure 10 is divided among five categories: work (the cost of the computation proper), filaments overhead, data transfer (page faulting and servicing), synchronization overhead (sending and receiving synchronization messages), and synchronization delay (differing barrier arrival times). A range is given for the times of the six interior nodes because of their different synchronization delays. The overall time is longer than that previously reported because this test was run using profiled code. Because the programs were started on each node in succession, and the system initialization and termination are included, the total time taken by each node in Figure 10 is not equal.

The Jacobi iteration times reported in section 4.2 take advantage of two performance enhancements: the implicit-invalidate page consistency protocol (PCP) and multiple pools. The communication overhead is

Nodes	1	2	4	8
DF Time (sec)	212	103	61.4	40.9
DF Speedup	1.01	2.09	3.50	5.26

Figure 11: Jacobi iteration, Write-Invalidate PCP, 256×256, $\epsilon = 10^{-3}$, 360 iterations. Sequential program: 215 sec.

Nodes	1	2	4	8
DF Time (sec)	212	104	65.5	48.5
DF Speedup	1.01	2.07	3.28	4.43

Figure 12: Jacobi iteration, Implicit-Invalidate PCP, single pool, 256×256, $\epsilon = 10^{-3}$, 360 iterations. Sequential program: 215 sec.

reduced by using the implicit-invalidate PCP, which has fewer messages than the write-invalidate PCP. The write-invalidate PCP requires invalidate messages to be sent, received, and acknowledged. The performance improvement, 3% and 6% on 4 and 8 nodes, can be seen by comparing the results in Figure 11 with those in Figure 5. The times for single-pool, non-overlapping Jacobi iteration are shown in Figure 12. Overlapping communication leads to a 9% and 21% improvement on 4 and 8 nodes, as can be seen by comparing the times in Figure 12 with those in Figure 5.

5 Related Work

There is a wealth of related work on threads packages. In general-purpose threads packages, such as uSystem [BS90], each thread has its own context. Consequently, the package has to perform a full context switch when it switches between threads. (This is true even in threads packages with optimizations such as those described in [ALL89]). The overhead of context switching is significant when threads execute a small number of instructions, as in Jacobi iteration. Hence, general-purpose threads packages are most useful for providing coarse-grain parallelism or for structuring a large concurrent system.

A few threads packages support efficient fine-grain parallelism, e.g., the Uniform System [TC88], Shared Filaments [EAL93] and Chores [EZ93]. The first two restrict the generality of the threads model. In particular, the Uniform System uses task generators to provide parallelism, much in the way a parallelizing compiler works, and in Shared Filaments a thread (filament) cannot block. On the other hand, Chores uses PRESTO threads as servers, so it can block a chore when necessary. However, Chores does not have a distributed implementation.

Support for the recursive programming style does not generally exist on distributed-memory systems. For example, in Munin [CBZ91], the user must program recursive applications using a shared queue (bag) of unexecuted tasks and must explicitly implement and lock the queue. DF allows recursive programs to be written naturally and efficiently by means of fork/join filaments.

Several systems use overlapping to mask communication latency. Active Messages [vCGS92] accomplishes overlap explicitly by placing prefetch instructions sufficiently far in advance of use that the shared data will arrive before it is used. On the other hand, DF achieves overlap implicitly through multithreading. Both require some programmer support to achieve maximal overlap of communication and computation. In Active Messages, the user has to make sure the prefetch is started soon enough, and in DF the user must correctly place filaments in pools (although we are working on making the placement automatic). The Threaded Abstract Machine, TAM [CGSv93], uses Active Messages to achieve overlap of communication and computation in a parallel implementation of a dataflow language.

CHARM is a fine-grain, explicit message-passing threads package [FRS+91]. It provides architecture independence, overlap of communication and computation, and dynamic load balancing. CHARM has a distributed-memory programming model that can be run efficiently on both shared- and distributed-memory machines. DF provides functionality similar to CHARM using a shared-memory programming model.

The Alewife [KCA91], a large-scale distributed-memory multiprocessor, provides hardware support for overlapping communication and computation. It provides the user with a shared-memory address space and enforces a context-switch to a new thread on any remote reference. The Alewife and DF use similar ideas, except that DF is a software implementation requiring no specialized hardware.

VISA, a DSM written for the functional language Sisal, also uses overlapping [HB92]. Suspended threads are pushed on a stack, so there can be many outstanding page requests. The disadvantage of a stack-based approach is that threads are resumed in the inverse order in which they request pages. In DF a thread is scheduled as soon as the page it requested arrives, whereas in VISA a thread is executed when it is popped off the stack. Thus, in VISA a faulting thread cannot execute until after the page it requested arrives and after completion of all other threads that were subsequently started.

False sharing occurs when two nodes access locations within the same page, and hence it can cause thrashing. Mirage uses the *time window coherence protocol* to control thrashing. In particular, a node keeps a page for some minimum time period to guarantee that it makes some progress each time it acquires the page [FP89]. Munin uses *release consistency* in the write-shared protocol to handle false sharing [CBZ91]. The memory is made consistent at synchronization points so there is no thrashing. TreadMarks provides lazy-release consistency on a network of Unix workstations [KDCZ94]. Simulation has shown that this greatly reduces the number of messages relative to the number of messages needed in systems like Munin. Some DSMs provide different granularities of memory consistency. For example, Clouds [DJAR91] and Orca [Bal90] provide shared memory objects. This provides consistency at the granularity of user-level objects instead of operating system pages, which can reduce thrashing. Blizzard [SFL+94] and Midway [BZS93] minimize false sharing by providing coherence at cache-line granularity. Midway keeps a dirty bit per cache line and propagates changes at synchronization points.

DF controls thrashing by using the Mirage window protocol and by providing the user control over the granularity of DSM pages. DF provides no support for concurrent writers to the same page; instead, the user must lay out the data such that thrashing will not occur. Currently, this must be solved at a higher level, such as by a compiler. We are investigating ways to simplify this task.

6 Conclusion

We have argued that fine-grain parallelism is useful and that it can be implemented quite efficiently on a workstation cluster without modifying either the hardware or the operating systems software. The Distributed Filaments package has a simple interface, and we have found it easy to use when programming parallel applications. DF achieves efficiency by overlapping communication with useful computation, by using a new datagram protocol that is both fast and reliable, and by automatically balancing the work generated by fork/join computations. The net result is that DF achieves good speedup on a variety of applications. In particular, as long as an application has a reasonable amount of computation per node, communication latency due to paging messages can be effectively masked. (However, if the cost of latency drops below that of context-switching, overlapping may no longer be beneficial.)

This paper describes DF, which is implemented on a cluster of Sun workstations running SunOS. We have prototypes running on a cluster of Sun workstations running Solaris, DEC workstations running Mach, and an Intel Paragon multicomputer running OSF. We are developing more application programs to provide a better basis for performance comparisons. We are also working on a number of improvements to the DF package itself: automatic clustering of filaments that share pages into execution pools, automatic data placement, experiments with different types of barriers for large numbers of processors, and support for explicit message passing as well as DSM.

Acknowledgements

Dawson Engler and David "the Kid" Koski provided numerous technical ideas and did lots of implementation work. Bart Parliman ported DF to the Mach operating system. Gregg Townsend provided comments on early drafts of the paper. The referees also made many constructive suggestions.

References

[ALL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Pursin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[Bal90] Henri E. Bal. Experience with distributed programming in Orca. *Proc. IEEE CS 1990 Int Conf on Computer Languages*, pages 79–89, March 1990.

[BS90] Peter A. Buhr and R.A. Stroobosscher. The uSystem: providing light-weight concurrency on shared memory multiprocessor computers running UNIX. *Software Practice and Experience*, pages 929–964, September 1990.

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

[CGL86] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. In *Thirteenth ACM Symp. on Principles of Programming Languages*, pages 236–242, January 1986.

[CGSv93] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM—a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, August 1993.

[CZ83] D.R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 128–140, October 1983.

[DJAR91] Partha Dasgupta, Richard J. LeBlanc Jr., Mustaque Ahmad, and Umakishore Ramachandran. The Clouds distributed operating system. *Computer*, pages 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. TR 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.

[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.

[Fre94] Vincent W. Freeh. A comparison of implicit and explicit parallel programming. TR 93-30a, University of Arizona, May 1994.

[FRS+91] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–193–II–201, Boca Raton, FL, August 1991. CRC Press.

[HB92] Matthew Haines and Wim Bohm. 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.

[KCA91] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 91–101, April 1991.

[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.

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

[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 (to appear)*, October 1994.

[SHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, July 1993.

[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.

[TL93] Chanramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[vCGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser. Active Messages: a mechanism for intergrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.