Efficient Support for Pipelining in Distributed Shared Memory Systems*

Karthik Balasubramanian
David K. Lowenthal
Department of Computer Science
University of Georgia
{kar,dkl}@cs.uga.edu

July 17, 2002

Abstract

Though more difficult to program, distributed-memory parallel machines provide greater scalability than their shared-memory counterparts. Distributed Shared Memory (DSM) systems provide the abstraction of shared memory on a distributed machine. While DSMs provide an attractive programming model, they currently can not efficiently support all classes of scientific applications. One such class are those with recurrences that cause dependencies across processors or nodes. A popular solution to such problems is to use *pipelining*, which breaks the computation into blocks; each processor performs the computation of a block, which enables the next processor in the pipeline to compute its corresponding block. Once the pipeline is filled, the computation of blocks proceeds in parallel. While pipelining is useful, it is not efficiently supported by current DSM systems.

This paper presents an approach to integrating pipelining into DSM systems. We describe our design and implementation of one-way pipelining in a DSM. The key idea is to retain the shared-memory model, but design the extensions such that the execution will mimic what would be done in an explicit message-passing program. We show that one-way pipelining is superior to the two most common ways to program pipelined applications, which are distributed locks and explicit matrix transposition. Finally, we show that one-way pipelining is competitive with a hand-coded, explicit message-passing program.

1 Introduction

The computational requirements of many scientific applications have created a need for high-performance parallel architectures. These architectures can be broadly categorized as *shared-memory multiprocessors* and *distributed-memory multicomputers*. Distributed-memory machines provide greater scalability and higher performance than their shared-memory counterparts. However, writing a parallel program for a distributed-memory machine is more complicated because it requires the use of explicit message passing to communicate between the processors. To the parallel programmer (or compiler), the shared-memory model is simpler than the distributed-memory model.

Distributed shared memory (DSM) systems strive to provide the best features of each of the two kinds of parallel architectures—the power and scalability of the distributed model with the

*This work was supported by National Science Foundation CAREER Grant CCR-9733063.

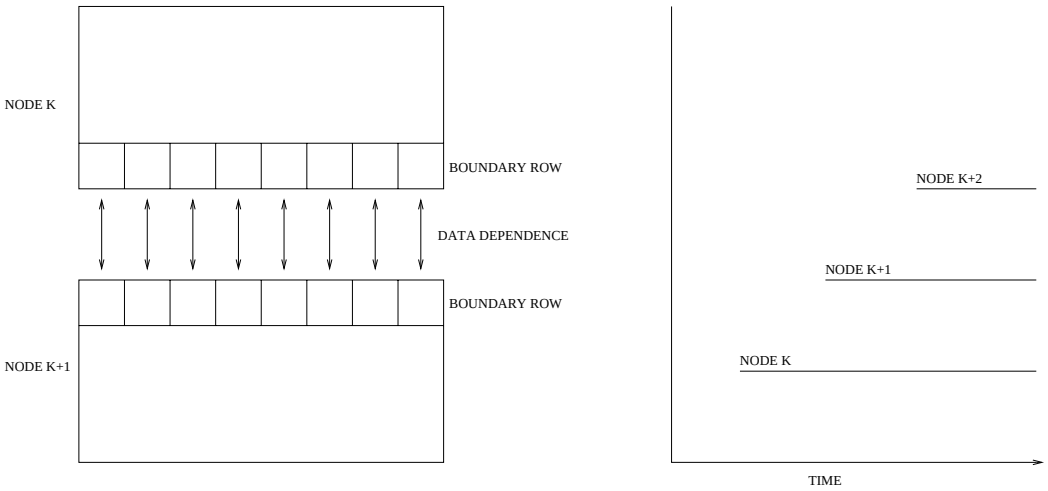

Figure 1: Computation with a cross-node dependence. Note that after the last node starts computation, all nodes run in parallel.

programming ease of the shared model. A DSM system provides the abstraction of the shared-memory model on a distributed-memory machine—that is, it eliminates explicit interprocessor communication. The model provided by the DSM system is similar to that of traditional virtual memory. The DSM provides a virtual address space, divided into pages, that is global to all processors (nodes). A reference by a node to non-resident data generates a *page fault* that is trapped by the DSM. The DSM pages the data into the node's memory by sending a message to the node that has the page. Hence, internode communication is done *implicitly* by the DSM system in a manner similar to how disk I/O is managed in paged operating systems.

Recently, DSM systems have received attention as attractive compiler targets[KT97, CDLZ97]. This is because it is much easier to generate code without having to determine at compile time whether data is local or not. Furthermore, when communication patterns cannot be determined at compile time, compilers often have to generate code with all to all communication, which is prohibitively expensive. On the other hand, with a DSM backend, communication is performed only on demand.

Unfortunately, current DSM systems are not universal enough to serve as general-purpose compiler backends. In particular, many scientific programs exhibit recurrences in their problem formulations. These recurrences give rise to *cross-node dependencies*, which result when, between synchronization points, an update to a data element on one (node) depends on the value of a data element on another (node). The dependence enforces an ordering on the execution of the loop as well as a message transfer between nodes. These dependencies tend to slow down parallel execution and sometimes even serialize the loop. This paper focuses on dependencies that arise from row sweeps over data arrays followed by column sweeps. These types of dependencies are quite common in scientific codes, including ADI integration, 2-d FFTs, and implicit hydrodynamics codes. A computation that has this kind of dependence is shown pictorially in Figure 1.

In a DSM system, one solution to cross-node dependencies is to compute the transpose of the matrix before the loop so that cross-node dependencies are eliminated. After the loop is executed, the matrix is transformed to its original state by computing another transpose. This solution, although simple to implement on a DSM system, is typically not viable due to excessive communication overhead. An alternative solution is to *pipeline* the loop. Software pipelines provide

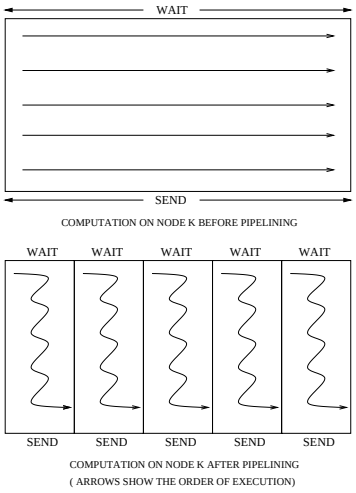

Figure 2: Pipelined computation. The computation proceeds by blocks instead of by rows.

an efficient solution in computations that exhibit recurrences that cause cross-node dependencies. Instead of proceeding by rows, the computation proceeds by blocks. This reduces the time that a node has to wait before it can begin the computation. Figure 2 pictorially shows a pipelined computation.

However, implementing explicit software pipelines in a DSM system is not straightforward. In particular, it requires (1) allowing nodes to communicate without sending explicit messages, (2) sending the minimum amount of data, and (3) avoiding thrashing, a well-known DSM problem.

This paper presents an approach to software pipelining for a DSM system. We have extended Filaments [FLA94, LFA96], a software kernel that provides a DSM, to support software pipelines. We provide the user (or compiler) with all functions needed to pipeline a program. Our implementation runs on a cluster of workstations connected by a dedicated Fast Ethernet. Performance results show that our one-way pipeline implementation achieves good speedup and is competitive (e.g., just over 10% slower on 8 nodes) with an explicit message-passing program, which is more difficult to write. Furthermore, our one-way pipeline is superior to using either locks or an explicit transpose, which are the two primary ways most programmers implement shared-memory pipelined programs.

The remainder of this paper is organized as follows. Section 2 gives an overview of the concepts and terminology relevant to the work presented in this paper. It also presents related work. Section 3 briefly describes the Filaments package. Section 4 discusses the implementation of one-way pipelines and compares it with distributed locks. Section 5 presents performance of both implementations and Section 6 concludes.

2 Overview and Related Work

In this section, we give an overview of concepts and terminology related to our work. We describe both DSMs and pipelining and then the difficulty involved in integrating the two. Finally, we present related work.

2.1 Distributed Shared Memory Systems

As discussed above, distributed-memory machines deliver high performance and scalability, but have a complicated programming model. A *distributed shared memory* (DSM) system provides an abstraction of shared memory on a distributed machine. The communication between nodes occurs implicitly using page faults. The basic principle behind a DSM system is an extension of virtual memory. The DSM provides a virtual address space that the nodes in the distributed system share. Application programs use this memory as conventional virtual memory. The DSM takes care of paging the data between the memories of the different nodes on demand.

Any node can access any location in the shared address space. A DSM system implements the mapping between the local memory and the shared address space, which is partitioned into pages; a memory reference by a node generates a page fault when the page is not resident. Such a *page fault* is trapped by the DSM system, which acquires a copy of the requested page by sending a message to the node that has the page.

A key issue involved in building a DSM system is solving the *memory coherence problem*; there are a number of strategies that can be used. These strategies are usually referred to as *page consistency protocols*; here, we discuss the write-invalidate protocol [LH89], where each page has either a single writer or multiple readers.

With write-invalidate, when a node incurs a page fault, the DSM moves a copy of the page to the node that faulted. If the fault is on a write, all other copies are invalidated, and the faulting node is given write permission. If the fault is on a read, the access permission becomes read-only on all copies (which may involve changing permissions).

Another key issue is avoiding thrashing, which occurs when a single page "ping-pongs" between two nodes. It is caused by false sharing, which occurs when at least two nodes have a copy of a page, and at least one is writing. Write-shared protocols [CBZ91] have been developed to tolerate thrashing in the case that there are no *true dependencies*, which occurs between two statements S_1 and S_2 when (1) S_1 precedes S_2 and (2) S_1 writes to a memory location that S_2 reads. Our applications all exhibit true dependencies, so we cannot use write-shared protocols.

2.2 Software Pipelining

Typically, in scientific applications, parallelism is extracted from loops because that is where most of the time is spent. Iterations of such loops are executed in parallel to achieve speedup over the sequential program. However, a number of these scientific applications exhibit recurrences that give rise to *data dependencies*. These dependencies enforce an ordering on the execution of the iterations in the loop where they occur, sometimes even forcing sequentialization of the execution of the entire loop.

Figure 3 shows a section of the Alternate Direction Implicit (ADI) integration program. Assume that the data (matrices) are distributed in groups of consecutive rows during the row sweep, which is what a typical parallelizing compiler would choose. This allows the row sweep to be executed in parallel without incurring any communication. However, in the column sweep, there is a dependence since the value of $X[i][j]$ depends on the value of $X[i-1][j]$. This means that node k must acquire a value from node $k-1$ in order to compute each value of its first row.

There are three primary solutions to this problem:

- serialize the second loop
- compute the transpose of the matrix before and after the column sweep phase and perform the second loop in parallel without any communication, or

```
  for iter := 1 to NUMITERS {
    /* row sweep */
    for i := 0 to N-1
     for j := 0 to N-1
       X[i][j] := F(X[i][j], X[i][j-1], A[i], B[i])
    /* column sweep */
    for i := 0 to N-1
     for j := 0 to N-1
       X[i][j] := F(X[i][j], X[i-1][j], A[i], B[i])
  }
```

Figure 3: Outline of sequential ADI program

```
for iter := 1 to NUMITERS {
  /* row sweep */
  for i := start to end
    for j := 0 to N-1
      X[i][j] := F(X[i][j], X[i][j-1], A[i], B[i])

  /* column sweep */
  for jj := 0 to N-1 by blocksize
    if (myId != 0)
      receive X[start-1][jj:jj+blocksize] from previous node
    for i := start to end
      for j := jj to jj+blocksize-1
        X[i][j] := F(X[i][j], X[i-1][j], A[i], B[i])
    if (myId != p-1)
      send X[end][jj:jj+blocksize] to next node
}
```

Figure 4: Pipelined, explicit message-passing version of ADI program. Variables `start` and `end` are local to each node and based on the number of participating nodes such that the work is divided evenly.

```
for iter := 1 to NUMITERS {
  /* row sweep */
  for i := start to end
    for j := 0 to N-1
      X[i][j] := F(X[i][j], X[i][j-1], A[i], B[i])

  /* column sweep */
  for jj := 0 to N-1 by blocksize
    if (myId != 0)
      LockAcquire(block[myProcessor][jj/blocksize]);
    for i := start to end
      for j := jj to jj+blocksize-1
        X[i][j] := F(X[i][j], X[i-1][j], A[i], B[i])
    if (myId != p-1)
      LockRelease(block[myProcessor+1][jj/blocksize]);
}
```

Figure 5: Pipelined shared-memory ADI program. Variables `start` and `end` are local to each processor and based on the number of participating processors such that the work is divided evenly. A two-dimensional array of locks is used for synchronization; an acquire is done before a block and a release is done afterwards.

- pipeline the computation, which avoids changing the data distribution.

Serializing the second loop is a simple solution but fails to extract parallelism. The second solution, computing the transpose, is highly inefficient; we show evidence of this in Section 5. The third solution, pipelining the computation, is the one that this paper discusses in detail. Software pipelining is a popular method to extract parallelism from problems that exhibit recurrences, such as the one shown in Figure 3.

Pipelined shared-memory and distributed-memory programs for ADI are shown in Figures 4 and 5. The computation is split up into blocks; each processor/node performs the computation blockwise. In the shared-memory program, synchronization is performed by acquiring and releasing appropriate locks; on the other hand, in the distributed-memory program, message receives and sends are done. This allows a a processor/node to start before the preceding processor/node has completed the computation on all its rows. Once the pipeline is filled, the computation of blocks on different processors/nodes proceeds in parallel.

2.3 The Problem

Although a pipeline can be naturally described in terms of explicit sends and receives, it is still simpler to program with shared memory. This is because no data need be sent; message passing is notoriously difficult to use correctly. (Furthermore, as discussed above, compilers can generate code much more easily for a shared-memory backend.) The shared-memory SUIF compiler [AALT95], for example, compiles codes like ADI integration into pipelined codes similar to the one shown in Figure 5.

As a DSM system provides the abstraction of shared-memory on a distributed machine, one

might expect it to be able to support programming pipelined applications with locks or semaphores. In other words, the program in Figure 5 might be expected to work correctly and efficiently on a DSM. Unfortunately, this is not the case. Programming pipelines in current DSMs as one would in a shared-memory multiprocessor can easily lead to (1) more than the minimum amount of data being sent, (2) thrashing, and (3) non-optimal (2-way) communication. To understand why, note that in a DSM system, the acquisition of a single block (likely much smaller than a page) will result in acquisition of an entire page. Furthermore, when a block is smaller than a page, we have a situation where two nodes are accessing different blocks, but both blocks are on the same page. Because one node is writing, thrashing could result. Finally, the DSM model is one of request/reply; a page fault causes a request, and the page is sent in reply. This is in contrast to the more efficient distributed-memory model, in which nodes simply block until they receive the boundary data from the previous node and then send the boundary data to the next node when they are done.

Our goal is to retain the DSM environment—no explicit message passing—while still sending the minimum amount of data and no extra consistency messages. This paper discusses the design, implementation, and performance of a DSM system that *efficiently* supports software pipelines.

2.4 Related Work

DSM Systems

There is an abundance of related work on hardware and software DSM systems (e.g. [BR90, KDCZ94, CBZ91, BZS93]).

Ivy, which was the first DSM, provides a shared virtual memory programming model on a network of workstations. Ivy's global address space is divided into virtual pages that correspond to physical pages. The memory mapping manager on each node views the physical memory on the node as cache of the shared virtual memory. Ivy uses a directory-based write-invalidate protocol to maintain memory consistency.

Munin [CBZ91] was a software DSM system that used a run-time system to address thrashing. They defined *release consistency* and a multiple-writer protocol. The release consistency model is relaxed compared to sequential consistency and allows multiple nodes to perform updates to a shared page without communication. Maintaining consistency is postponed until a synchronization operation occurs. This solves the problem of thrashing and, as long as there are no true dependencies, reduces traffic caused by thrashing.

Pipelining

Pipelining computations can provide an efficient solution to cross-node dependencies in loops. As a result, a number of parallelizing compilers and systems use pipelining to exploit parallelism. There is a wealth of related work on pipelining [AK87, BCG+95, HLKL97, KTG95, Lov77, VSM96]; this section presents some of it.

Before support for pipelining was built into parallelizing compilers, a number of researchers wrote programs that were pipelined by "hand". The computations in these application programs were pipelined by explicitly passing messages.

An optimizing Fortran D compiler developed at Rice University [Tse93] uses a number of optimization techniques to generate efficient parallel programs. This compiler analyzes programs, performs optimizations and generates code. The compiler uses optimization techniques that are guided by data dependencies. One of the optimization techniques used by the compiler to exploit parallelism is pipelining. The compiler applies mechanisms such as loop fusion, loop interchange and strip mining to pipeline the computation of the loop. Two of the techniques that the compiler

uses to exploit pipeline parallelism are fine- and coarse-grain pipelining. With fine-grain pipelining, loop interchange is used to minimize the granularity of the pipeline, maximizing pipeline parallelism at the cost of increasing the communication overhead. Coarse-grain pipelining attempts to balance the parallelism with communication overhead by using loop transformations to increase the granularity of the pipeline.

PARADIGM [BCG+95] is a parallelizing compiler built for distributed-memory multicomputers. PARADIGM performs a number of optimizations to generate efficient parallel code. It uses coarse-grain pipelining to exploit parallelism in loops with cross-node dependencies. The system uses an execution time estimate to compute the total cost of execution of the loop. This total cost is then minimized with respect to the block size to select the granularity for the pipeline. Once this granularity has been calculated, various loop transformations are used to change the granularity of the pipeline to improve execution time.

The block size can be crucial to the performance of a pipelined parallel program. A smaller block size results in smaller idle time per node, but increases the communication overhead. A larger block size has the reverse effect—it reduces the communication overhead, but results in a larger pipeline fill time. The ideal block size is one that will minimize the completion time. A run-time system [LJ99] has been developed to analyze and choose the best block size for the pipeline. The system analyzes the program on the first iteration and based on timing results, it chooses an effective block size (which can be nonuniform) for the computation. The best block size found is then used for the remainder of the computation.

3 Filaments

Filaments is a software kernel that efficiently supports a fine-grain, shared-memory programming model on a range of architectures. It includes very lightweight threads, called filaments, as well as a multithreaded distributed shared memory (DSM) and an efficient communication protocol. This section describes only those parts of Filaments relevant to this paper, which are the DSM and reliable UDP communication protocol.

3.1 Filaments Distributed Shared Memory System

The DSM in Filaments is implemented entirely in software. Hence, it is inexpensive and extremely flexible. With a relatively small amount of operating system support, it can be ported to different hardwares and operating systems. The Filaments software DSM is already supported on SPARC/Solaris, PentiumPro/Solaris, and PentiumPro/Linux.

The DSM divides the address space of the node into two sections—a shared section and a private section. The shared section, currently divided into 4K pages, contains the user data that is shared by all the nodes. The private section contains the data that is local to a node; for example, program code, local variables in the program, etc.

There are two events that could occur in the DSM system: remote page fault and message pending. A message pending event is generated when a node receives a message. It is handled asynchronously by an event handler routine that is called when a `SIGIO` signal is generated (by the incoming message). A remote page fault occurs when a thread on a node accesses a remote page.

The pages in the DSM are protected using the `mprotect` call. When a thread tries to access a remote page, a `SIGSEGV` signal is generated. A signal handler traps this signal and sends out a request for the remote page and also suspends the thread. It then calls the scheduler, which will execute another thread. When the remote page request is satisfied, the scheduler places the faulting thread in the ready queue so that it can be executed again. Since there are a number of threads

that can be executed, the DSM system can support several outstanding page requests. Indeed, the ability to overlap communication with computation is the main difference between Filaments and other software DSMs.

Filaments can support several *page consistency protocols*; currently, it supports *migratory, write-invalidate and implicit-invalidate*. The migratory protocol keeps exactly one copy of each page. The page moves from node to node when necessary. With write-invalidate, multiple read-only copies of a page are allowed, but the copies are invalidated explicitly when one of them is written. Implicit-invalidate is a more efficient version of the write-invalidate protocol. With implicit-invalidate, all read-only copies of a page are implicitly invalidated at each synchronization point. Hence, explicit invalidate messages are unnecessary. However, implicit-invalidate works only with regular problems with stable data sharing patterns. In this paper we focus on (and modify) the write-invalidate protocol.

3.2 Communication Model

A DSM requires reliable communication as well as low overhead. Filaments uses the User Datagram Protocol (UDP) which is an unreliable but fast protocol. On UNIX, the choice is between TCP and UDP. Although TCP is reliable, it does not scale well with the number of nodes. UDP is unreliable but does scale well. Additionally, it is slightly faster than TCP and also supports broadcast messages. Filaments, therefore, uses UDP with reliability built on top of it. The result is a fast and reliable protocol.

There are two kinds of messages: request and reply. A node sends out a request to which another node replies. If the request is lost or delayed, the sending node retransmits the request. This means that the each request message sent has to be buffered. However, the request messages are small, so buffering is not expensive. The request message is added to a list of sent messages and each time a reply is received, the corresponding request message is removed from the list. Our version of UDP does not buffer DSM page data. Instead, only the actual reply is buffered and the data part of the reply is constructed from the contents of the page at the time of retransmission. This is possible because all nodes wait at synchronization points until all outstanding requests are satisfied, which in turn means that if the program has no race conditions, a request is always satisfied by a reply with a consistent copy of the page [FLA94].

4 Implementation

This section presents the implementation of pipelining in a DSM. We implemented one-way pipelines that are written using a shared-memory model, while the execution model is that of distributed-memory. For comparison, we also implemented distributed locks (which is the traditional "shared-memory" way to pipeline). First, we discuss elements common to both implementations. Then, we discuss the user interface and implementation of one-way pipelining. Note that we initially assume only one array and an upward dependence; later, we relax these restrictions. Finally, we discuss the user interface and implementation of distributed locks.

4.1 Common Elements

The simplest solution to DSM pipelining would be to force a node to wait, with some blocking primitive, until the previous node has finished with the computation. Then we could let the DSM acquire the data implicitly by faulting on the corresponding block of data from the preceding node's last row. However, there are three problems with this approach. First, as each last (bottommost)

row on a node is referred to by *two* nodes, the page(s) on which the last row is located will be transferred between the two nodes a large number of times. For example, suppose the last row takes one page and is divided into $n > 1$ blocks; consider the first two nodes (which share a row). After computation of the first block on the first node, the second node would fault on the entire page. Hence, we don't have the minimum amount of data being transferred, because we need a page fault on the (same) page for *each* of the n blocks. Second, we have two nodes accessing the page, and one writing. This situation is well-known to cause thrashing. Note that using a write-shared protocol is not possible here, because there is a true dependence. (Write-shared works only when the concurrent accesses are to completely disjoint memory locations.) Finally, two-way communication is necessary when using traditional DSM protocols.

The solution to the first two problems, for both one-way pipelines and locks, requires a new PCP for Filaments that we call *pipeline-invalidate*. Pipeline-invalidate is the same as write-invalidate except that it marks the last row of each node as `READ/WRITE`. This ensures that a node will not generate a page fault when it refers to the data on the last row of the preceding node. Note that with pipeline-invalidate, the data now has to be explicitly sent to the node requiring it as there will be no page fault on boundary rows. Further, synchronization must be inserted to ensure that computation on a node begins only after the appropriate data has been received. There are two different ways to provide this synchronization: *one-way pipelines* and *distributed locks*. The remainder of this section describes the implementation of these two approaches in detail. Each approach has two subsections: one describing the user interface and the other describing the implementation.

4.2 One-Way Pipelines

4.2.1 User Interface

Our modified Filaments DSM provides the user with a simple API for implementing a pipelined program. The API consists of `InitPipeline`, `WaitBlock`, and `BlockDone`. The user creates a pipeline for an array using the `InitPipeline` function call. The parameters passed to this function are:

- id of pipeline
- number of blocks
- size and start address of array

The function `WaitBlock` forces a node to wait for a block of data from the preceding node and returns when the DSM receives the data from the preceding node. All nodes except the first call `WaitBlock`. The function `BlockDone` marks a block of data as done, which notifies the Filaments DSM that it can be sent to the next node in the pipeline. It is asynchronous and called by all nodes except the last. Note the *DSM* deals with the details of receiving and sending, not the programmer (or compiler). Both functions have a single argument, which is the id of the pipeline. This is actually a *simpler* programming model than the shared-memory program shown in Figure 5, because our system keeps the state of the pipeline. The following section describes the implementation of the one-way pipelining.

4.2.2 Implementation

The one-way pipeline consists of a list of structures, each of which represents a pipeline for a matrix (see Figure 6). The list structure of the pipeline provides support for multiple pipelines with different matrices and different block sizes. The pipeline structure itself contains the following

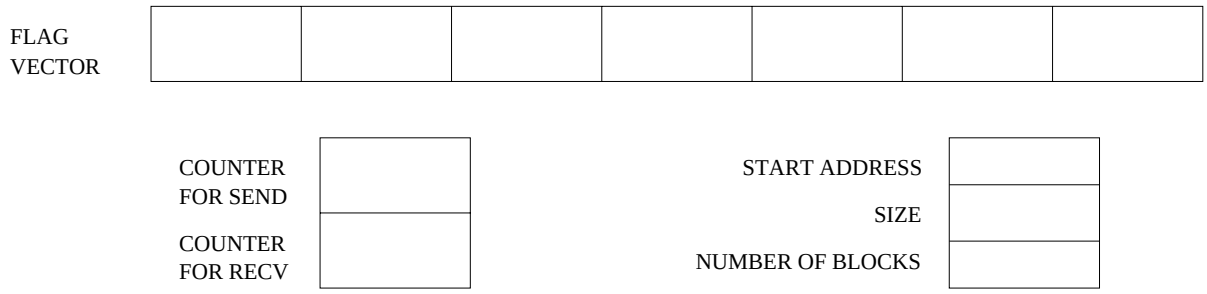

Figure 6: One pipeline on a node. In this example there are seven blocks. There is a flag vector to keep track of potentially multiple messages, and separate counters indicating which block is next to send and receive, respectively.

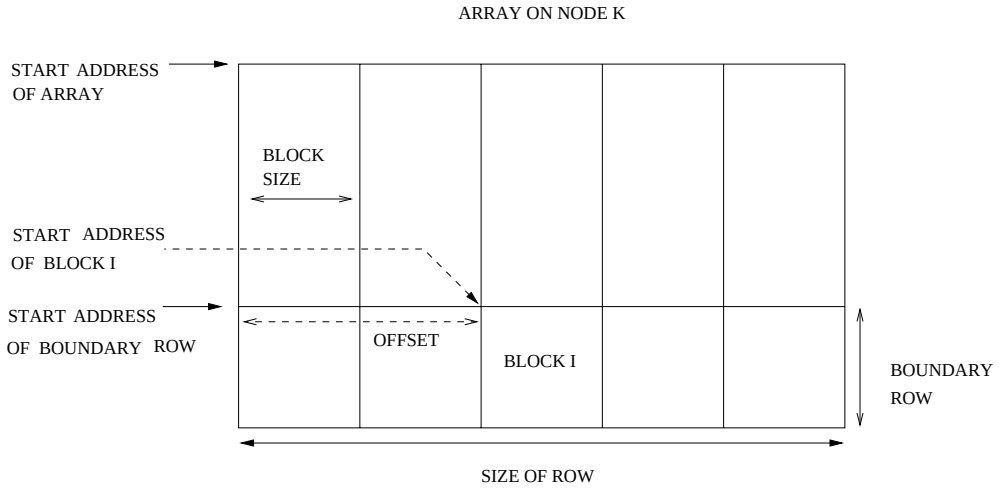

Figure 7: Block address calculation.

information: two counters that specify the block that needs to be sent/received next, a flag vector that tells the node if the block of data has been received, and all the information needed to send/receive the data to/from another node (the start address of the matrix, size of the matrix, and number of blocks). A node that is waiting for a block spins on the corresponding flag, which is set when the message containing the appropriate block is received. The flag vector acts as the synchronization variable for the pipeline. A vector is needed because a node may receive arbitrarily many blocks of data before it completes the computation on the current block; each block must be separately accounted for. Each pipeline is associated with a single matrix and is indexed sequentially by an id in the list of pipelines. A call to `InitPipeline`, described above, creates a pipeline and appends it to the list. The arguments, with the exception of the id, are used in sending and receiving data.

Function `WaitBlock` spins on the flag for the block of data that the node is waiting for from the preceding node. The argument passed to this function is the id of the pipeline associated with the matrix. The flag is reset when the block of data arrives from the node that completed the computation on that block.

Function `BlockDone` causes the Filaments DSM to send the block whose computation was just

completed to the node below, along with the block number and pipeline id. To determine the precise data to send, the address of the start of the block has to be calculated before sending the data, and is done as follows (see Figure 7): the page number of the last row is first computed using the start address of the matrix and its size. The address of this page is obtained from the page table. The offset of the block within the last row is calculated using the block number and the size of the block. The start address of the block is then calculated by adding the block offset to the start address of the last row's page.

An incoming message containing a block of data is processed by a specialized handler function, which reads in the message and, using the id of the pipeline stored in the header of the incoming message, traverses the list of pipelines to obtain the pointer to the correct pipeline. The start address of the block that has been received is calculated in the same manner as the `BlockDone` function described above. The data is then copied into the correct address and the flag for that block is reset, allowing the node to begin the computation.

Improvements

In order to extend the pipeline to handle dependencies on rows above *and* below (the example presented in Chapter 2 had only an upward dependence), we actually provide four API functions on blocks: `WaitBlockAbove`, `BlockDoneBelow`, `WaitBlockBelow`, and `BlockDoneAbove`. The first two are the functions described earlier as `WaitBlock` and `BlockDone`. Function `WaitBlockBelow` is the same as `WaitBlockAbove`, except that it waits on blocks from nodes below. Function `BlockDoneAbove` is the same as `BlockDoneBelow`, except that the DSM is informed to send boundary data up. Further, the pipeline-invalidate PCP had to be modified to support the bidirectional pipeline by marking the first as well as the last row of each matrix as `READ/WRITE`. Finally, we add a separate send and receive counter, as well as a separate flag vector.

In a number of applications (including ADI integration), more than one matrix is involved in the computation within the same loop(s). In such cases, creating a pipeline for each matrix will cause additional overhead due to an increase in the number of messages sent and received. To decrease this overhead, we improved the implementation so that each pipeline could handle more than one matrix. This means that each message sent contains data from more than one matrix. We implemented a scatter/gather approach that packs one block of data from each of the matrices in the pipeline into one message. On reception of the message, the receiving node unpacks the data and copies the data block of each matrix into the correct location. This approach significantly reduces the number of messages sent and received when multiple matrices are involved in the computation. The following section describes the second approach to adding support for pipelined programs, which is *distributed locks*.

4.3 Distributed Locks

4.3.1 User Interface

The Filaments API for Distributed Locks provides the user with three functions that can be used to pipeline the computation: `CreateLock`, `LockAcquire`, and `LockRelease`. The `CreateLock` function enables the user to create a lock and associate it with a block of data to protect. The arguments passed to this function are:

- the id of the lock
- the size and start address of the block of data associated with it
- the owner of the data

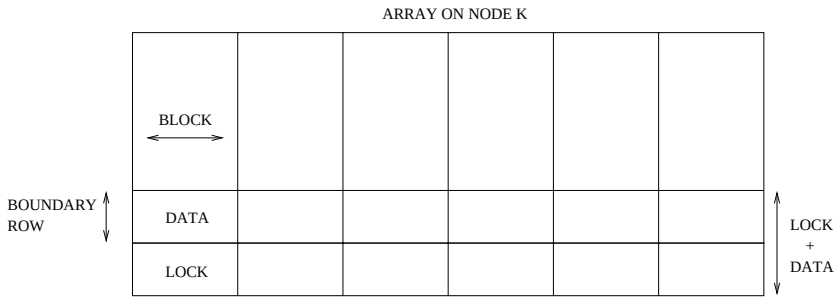

Figure 8: Distributed lock structure. Note the data is bound to the lock.

Functions `LockAcquire` and `LockRelease` take a single parameter, which is the lock id. A call to `LockAcquire` function attempts to acquire a lock. If the lock is held by another node, the node blocks, waiting for the lock to be released. The `LockRelease` function releases the lock held by a node so that it can be acquired by other nodes requesting it.

4.3.2 Implementation

The lock-based approach is built on a `request/reply` structure as opposed to the one-way pipeline, which employs unidirectional communication. With the lock-based approach, each node has to acquire a lock before it can begin the computation on a block of data. The node releases the lock once it is done with the computation.

However, the locks that we implemented are different from conventional locks in that each lock is bound to the data that it protects (see Figure 8). This is an idea borrowed from the Midway DSM system [ZSB94] and is essential for better performance because the data is distributed. If the lock and the data are not associated, separate messages for acquiring the lock and faulting on the data would be required. By binding the lock to the data it protects, a node that acquires the lock will acquire the associated data in the same message.

The lock data structure itself contains the following data:

- lock id
- spin variable
- start address and size of the data block it is bound to
- current owner of the lock
- the wait queue

The owner field identifies the node that currently possesses the lock. The wait queue holds outstanding lock acquire requests. The start address and data size are used to determine which data to send, and the spin variable is true if the lock is held.

A call to the `CreateLock` function creates and initializes a lock. The parameters passed to this function are the values of the various fields of the lock data structure. Each lock is assigned a unique id by which it is identified. However, assigning unique id's to the locks is a non-trivial problem since the locks are distributed and we must assign id's to the locks in a consistent manner. (We could put the locks in DSM space, but this can cause thrashing; our goal is to make the locks as efficient as possible.) A lock shared by two nodes must have the same id on both. Figure 9 represents the

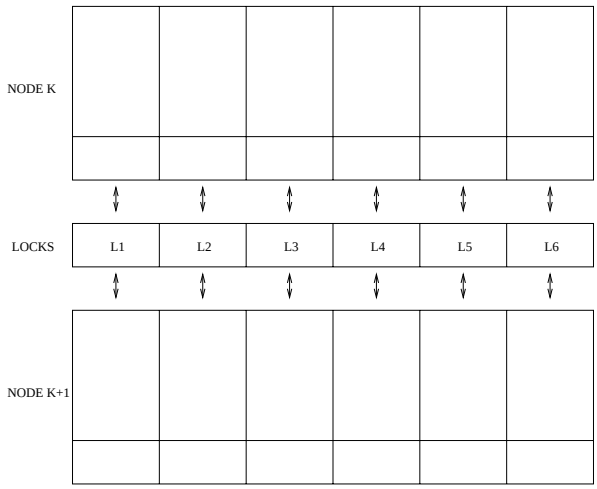

Figure 9: Lock sharing between nodes. Each non-boundary node shares a set of locks with the node above and the node below.

logical view of lock sharing between two nodes. If we assign the id for locks automatically, we must keep track of the ids assigned on one node to ensure that another node does not reassign an id to a different lock. For this reason, we chose to let the user (or compiler-generated) code assign the lock id by passing it to the `CreateLock` function. In this manner, the consistency among the distributed locks can be maintained in a simple way.

A node must acquire the lock associated with a block of data before it can begin computation on that block. Each node acquires the lock by invoking the `LockAcquire` function. The `LockAcquire` function uses the lock id parameter passed to it to index into the array of locks. If the owner of the lock is the node itself, then the node simply sets the spin variable to true and begins computation. However, if the node is not the owner of the lock, it sends a lock acquire message to the current owner of the lock and spins, waiting for notification of the release of the lock.

The handler function invoked on the receipt of a lock acquire message inspects the value of the spin variable for that lock. If the value of the spin variable is false, the handler transfers ownership of the lock and piggybacks the data block in the message. The actual transfer of the data is similar to the data transfer in the one-way pipeline approach. If the value of the spin variable is true, then the node holds the lock and has not completed the computation of the block. In this case, the handler enqueues the request; then, it can either just continue, or it can send a denial message. We tried both; in our case the latter approach was better because the Filaments reliable UDP communication protocol package continually retransmits messages if it gets no reply.

The handler function for receiving the lock and data simply copies the data into the correct location and resets the current owner of the lock to itself. Again, the unpacking of the data is done in the same manner as in the implementation of the one-way pipeline.

The `LockRelease` function releases the lock held by the node for the block of data on which the computation was completed. If the wait queue of the lock is empty, the function simply sets the spin variable to false, signifying that the lock is free. If the wait queue is not empty, then the request at the front of the queue is retrieved and the data and ownership of the lock are sent to the appropriate node.

```
ADI:

for(i=1; i<n-1; i++) {
  for(j=0; j<n; j++){
    b[i][j] = b[i][j] - (1.0 / b[i-1][j]);
    uh[i][j] = uh[i][j] + (uh[i-1][j] / b[i-1][j]);
    u[i][j] = b[i][j] + u[i-1][j];
  }
}

Hydro:

for(i=0; i<n-1; i++)
  for(j=0; j<n; j++)
    A[i][j] = (A[i][j-1]*B[i][j] + A[i][j+1]*C[i][j]
              +A[i-1][j]*B[i][j] + A[i+1][j]*C[i][j])/4;
```

Figure 10: Outlines of test programs

5 Performance

This section presents the performance of ADI integration and an implicit hydrodynamics kernel using one-way pipelining, distributed locks. We also present performance of an explicit array transpose.

First, we present a description of the two test programs and the experimental environment. Next, we present experiments that show how we determined the best block size. The subsequent sections present the timing results of the various tests we ran and shows that the performance of one-way pipelining is superior to that of distributed locks. We also compare the one-way pipelined programs to hand-coded, explicit message-passing programs and show that the former are competitive with the latter. Finally, we present the timing results of the transpose operation and show that the one-way pipeline provides a much more efficient solution.

5.1 Application Descriptions and Experimental Environment

Our two test programs are ADI integration and a hydrodynamics kernel. Both programs consist of a prespecified number of iterations, where each iteration performs an update of each point in a two dimensional matrix. In Hydro, each update is based on the values of the four points surrounding it. In ADI, each update depends on the value of a point on the previous row. Clearly, both these applications have cross-node dependencies, which prevent parallelization of the computation without intra-iteration communication. The sections of code that perform the update are shown in Figure 10. Note that each test has multiple matrices and that Hydro has both upward and downward dependencies; hence, they require the improvements described in Section 4.

The input parameters to the test programs are matrix size, block size, and number of iterations. We ran a number of tests, all compiled with `cc -O`, with various combinations of the input parameters. All programs executed for 100 iterations. The tests were run on a network of Pentium

Nodes	Array Size	Block size	ADI (sec)
2	256	16	9.18
2	256	32	9.12
2	256	64	9.83
4	256	16	8.19
4	256	32	7.33
4	256	64	7.83
8	256	1	44.2
8	256	16	9.14
8	256	32	6.10
8	256	64	7.67
8	256	256	16.1

Table 1: Effect of block size on performance of ADI, size 256.

Pro workstations connected by a dedicated 100 Mbps Fast Ethernet. Each workstation has a 200 MHz processor running Solaris, a 64K mixed instruction and data cache, and 64 megabytes of main memory. The network is isolated so that no outside network traffic interferes with the executing programs. Each test was run at least three times and the reported result is the median.

5.2 Determining Block Size

The block size in pipelined code has a crucial effect on performance. The larger the block size, the smaller the number of messages sent over the network. However, the larger the block size, the longer the latency. The time taken to fill up the pipeline is the sum of the block computation times for the first block and the message latency on nodes 1 through $n-1$. In other words, it is the time that the last node has to wait before it can start on the computation. A block size of n, where n is the size of the matrix, sequentializes the computation, since no node can start computation until any preceding node has completely finished its computation. With a block size of 1, there is a message sent out for each point. In this case, the communication overhead causes the performance to degrade to worse than sequential. The ideal block size is a trade off between the communication cost and the pipeline fill time. Table 1 shows a range of times for various array and block sizes. For both applications, a block size of 32 was experimentally determined to be the most effective (out of all powers of two).

5.3 Performance of One-Way Pipeline and Distributed Locks

Tables 2 and 3 show the timing results of the tests using one-way pipelining and distributed locks. They show that the one-way pipeline allows good speedup and distributed locks achieve modest speedup. On smaller matrix sizes, the speedup tapers off as the number of nodes increases from 2 to 8. On larger sizes, the increased ratio of computation to communication allows for better speedup. In particular, with ADI using one-way pipelining, the speedup obtained on 8 nodes with a matrix size of 256 is only 2.51, whereas the speedup with a matrix size of 1024 is 5.7. However, for distributed locks, the best speedup for ADI was was less than 4. The better performance of one-way pipelining was similarly evident on Hydro. Note in general that a larger matrix size also tends to amortize the effect of the pipeline latency.

Nodes	Array Size	Block size	ADI (sec)	Hydro (sec)
1	256	32	15.2	6.94
1	512	32	64.7	28.3
1	1024	32	265	113
2	256	32	9.12	4.71
2	512	32	35.3	17.3
2	1024	32	139	71.1
4	256	32	7.33	4.20
4	512	32	21.4	12.3
4	1024	32	75.6	40.5
8	256	32	6.10	3.73
8	512	32	16.9	13.4
8	1024	32	46.4	30.6

Table 2: Performance of one-way pipeline on ADI and Hydro for three different sizes. All tests were run for 100 iterations.

For comparison, the results of the explicit message-passing ADI program with size 1024 are shown in Table 4. This hand-coded program uses sockets for explicit communication instead of using implicit communication; otherwise the programs are the same. It shows that the one-way pipeline program is competitive; in the worst case, it is 11% slower than the message passing program. The lock-based pipeline program, on the other hand, is almost twice as slow. One should keep in mind that the one-way pipeline program is written with a shared-memory model, and so a slight degradation in performance compared to an explicit message-passing program is expected (for example, in one-way pipelining we must send explicit acknowledgments).

5.4 Locks vs. One-Way Pipeline

The results show that the lock-based solution is less efficient than the one-way pipeline, because of the higher overhead and increased number of messages (twice as many) of the former. The extra messages are due to the request/reply structure of the lock based solution, where each node has to first send out a message to acquire the lock and data. On the other hand, the one-way pipeline has no request messages; each node instead blocks until the data is received from the preceding node. One might think that the extra messages in the lock-based program do not contribute to the overhead since they are just requests and carry no data. However, we found by experimentation that there is little difference in sending an empty message than in sending a message with a small amount of data.

Furthermore, a node that receives a lock acquire message during its computation has to context switch to the handler function, process the message and then continue the computation. This is an added overhead in the lock-based solution. In fact, profiling showed that the function that performs the (identical) computation took longer when using locks than when using the one-way pipeline. All of these factors contribute to the superiority of the one-way pipeline to distributed locks.

Nodes	Array Size	Block size	ADI (sec)	Hydro (sec)
1	256	32	15.2	6.94
1	512	32	64.7	28.3
1	1024	32	265	113
2	256	32	10.5	6.00
2	512	32	38.3	20.7
2	1024	32	168	75.0
4	256	32	9.56	5.88
4	512	32	34.2	18.5
4	1024	32	133	50.7
8	256	32	8.52	7.30
8	512	32	21.7	17.2
8	1024	32	75.5	42.1

Table 3: Performance of lock-based pipeline on ADI and Hydro for three different sizes. Note the sequential times are the same as for the one-way pipeline results.

Nodes	ADI (sec)
1	265
2	137
1	73.2
1	41.9

Table 4: Performance of explicit message passing program on ADI (1024×1024, block size 32, 100 iterations).

5.5 Transpose

As we discussed in Chapter 2, an alternate solution in some problems is to perform each phase in parallel without any communication, with a transpose in between the two phases. However, computing the transpose of a distributed matrix is an expensive operation because it involves all to all communication between the nodes. To illustrate this, we present the timing results we obtained for a DSM transpose of matrices of different sizes in Table 5. We found that the times taken for a single DSM transpose is greater than the time taken by the pipelined versions of the ADI and the Hydro programs for 1 iteration (which are presented in Table 6).

In addition, we will have to compute the transpose twice in an application such as the one in Section 2, because we will have to compute the transpose of the matrix again at the end of the second phase to return the matrix to its original state. From the data presented in Tables 5 and 6, it is obvious that the DSM transpose is a much less efficient solution than the pipeline. For example, on the two-node test of ADI run with a matrix size of 256, we find that the pipelined program takes 0.058 sec per iteration. The DSM transpose takes 0.33 sec for a matrix of the same size. Noting that two transposes will be required by the program, we find that for Hydro, the transpose needs to be at least 10 times faster for it to compete with the performance of the pipelined version of the program. Note that one could write an explicit message-passing transpose. However these

Nodes	Array Size	Time (sec)
2	256	0.33
2	512	1.37
2	1024	4.49
4	256	0.36
4	512	1.43
4	1024	3.72
8	256	0.45
8	512	1.72
8	1024	4.26

Table 5: Timing results of DSM transpose.

Nodes	Array Size	Time – Hydro (sec)	Time – ADI(sec)
2	256	0.058	0.547
2	512	0.210	2.24
2	1024	0.815	11.2
4	256	0.039	0.346
4	512	0.130	1.36
4	1024	0.453	5.85
8	256	0.029	0.330
8	512	0.094	1.26
8	1024	0.279	4.99

Table 6: Single iteration times of one-way pipeline on ADI and Hydro.

are difficult to implement. Furthermore, they are not likely provide a ten-fold improvement (we tried one efficient algorithm, and it was no more than twice as fast). Hence, for our applications, pipelining the computation provides a more efficient solution than a transpose-based solution.

6 Conclusion

This paper discusses the issues involved in the implementation of an efficient pipelining in distributed shared memory (DSM) systems. We have presented the design and implementation of both one-way pipelining and distributed locks as extensions to Filaments, a software kernel that provides a DSM. We showed that one-way pipelining is the superior solution because it sends close to the minimal amount of data—in fact, it sends almost the same amount of data one would send in an equivalent program that uses message passing. For this reason, one-way pipelining is very competitive with explicit message-passing programs; the former was always within 11% of the latter in our experiments. Furthermore, we showed that one-way pipelining is far superior to using an explicit transpose. Indeed, our implementation of one-way pipelining increases the number of applications that can be efficiently supported by DSM systems.

References

[AALT95] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau-Wen Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[AK87] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *TOPLAS*, 9(4):491–542, October 1987.

[BCG+95] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[BR90] Roberto Bisiani and Mosur Ravishankar. Plus: A distributed shared memory system. In *17th Annual International Symposium on Computer Architecture*, pages 115–124, May 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.

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

[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.

[HLKL97] A.R. Hurson, Joford T. Lim, Krishna M. Kavi, and Ben Lee. *Parallelization of DOALL and DOACROSS Loops — A Survey*, volume 45, pages 53–103. Academic Press Ltd., 1997.

[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.

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

[KTG95] V.P. Krothapalli, J. Thulasiraman, and M. Giesbrecht. Run-time parallelization of irregular DOACROSS loops. In *Proceedings of Irregular '95*, pages 75–80, 1995.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37:41–54, November 1996.

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

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

[Lov77] D.B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):129–138, January 1977.

[Tse93] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[VSM96] Rob F. Van der Wijngaart, Sekhar R. Sarukkai, and Pankaj Mehra. The effect of interrupts on software pipeline execution on message-passing architectures. In *Proceedings of ACM Int'l. Conference on Supercomputing*, May 1996.

[ZSB94] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for distributed shared memory. In *First Operating Systems Design and Implementation*, November 1994.