RICE UNIVERSITY

Compiling for Software Distributed-Shared Memory Systems

by

Kai Zhang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. John Mellor-Crummey, Chairman
Senior Faculty Fellow
Computer Science

Dr. Alan Cox
Associate Professor
Computer Science

Dr. Ken Kennedy
Ann and John Doerr Professor
Computer Science

Dr. Rob J. Fowler
Research Scientist
Computer Science

Houston, Texas

October, 1999

Compiling for Software Distributed-Shared Memory Systems

Kai Zhang

Abstract

In this thesis, we explore the use of software distributed shared memory (SDSM) as a target communication layer for parallelizing compilers. For SDSM to be effective for this purpose it must efficiently support both regular and irregular communication patterns. Previous studies have demonstrated techniques that enable SDSM to achieve performance that is competitive with hand-coded message passing for irregular applications. Here, we explore how to effectively exploit compiler-derived knowledge of sharing and communication patterns for regular access patterns to improve their performance on SDSM systems.

We introduce two novel optimization techniques: compiler-restricted consistency which reduces the cost of false sharing, and compiler-managed communication buffers which, when used together with compiler-restricted consistency, reduce the cost of fragmentation. We focus on regular applications with wavefront computation and tightly-coupled sharing due to carried data dependence. Previous studies of regular applications all focus on loosely-coupled parallelism for which it is easier to achieve good performance. We describe point-to-point synchronization primitives we have developed that facilitate the parallelization of this type of applications on SDSM.

Along with other types of compiler-assisted SDSM optimizations such as compiler-controlled eager update, our integrated compiler and run-time support provides speedups for wavefront computations on SDSM that rival those achieved previously only for loosely synchronous style applications. For example, we achieve a speed up of 11 out

of 16 for SOR benchmark – a tightly-coupled computation based on wavefront, of a problem size of 4Kx4K. which compares favorably with the 14 out of 16 speed up which we obtain for Red Black SOR, – a loosely-coupled computation, of the same problem size under the same hardware and software environment. With the NAS-BT application benchmark using the Class A problem size, we achieved an impressive boost of speedup, from 4 out of 16, to 10 out of 16, on SDSM as a result of the compiler and runtime optimizations we described here.

Acknowledgments

There are several people I wish to thank for their assistance. Obviously, the person most responsible for this document other than myself is my advisor, John Mellor-Crummey, to whom I wish to express my sincere thanks for his kind and patient support, as well as for his direct contributions. The project could not have been so successful without the advice and suggestions provided by the other members of my committee. Many thanks should go to Guohua Jin for his joint work in shaping the shared memory dHPF compiler. I also wish to thank my friends and fellow students for their encouragement and help in the past three years. Special thanks to Iva Jean Jorgensen for pointing out many of my grammatical and spelling errors in this document. At last, I might have been unable to do this without the support of my loving family.

Contents

Tables

Illustrations

Chapter 1

Introduction

1.1 Motivation

Software distributed shared memory (SDSM) run-time systems support a shared-memory communication abstraction on message-passing multicomputers. A shared-memory communication layer provides a flexible base for developing parallel applications. Shared memory is a particularly convenient paradigm for irregular applications because its dynamic resolution of communication greatly simplifies access to and management of irregularly shared data. A previous study by Cox *et al.* [24] of two irregular applications showed that their performance using a SDSM communication layer exceeded that of compiler-generated message passing versions of these programs by 38% and 89%. This study also compared the performance of these SDSM implementations of irregular applications with hand-coded message passing counterparts and showed that the SDSM versions only lagged hand-coded performance by 7.5% and 16% respectively.

The aforementioned study also compared the performance of four regular codes on SDSM with both compiler-generated and hand-coded message-passing implementations. The SDSM parallel implementations of these regular codes (Jacobi, shallow, MGS, and 3D-FFT) lagged their message-passing counterparts by 5.5–49% [24]. A prior study by Dwarkadas, Cox, and Zwaenepoel that included other regular applications found that SDSM versions using compiler-directed runtime support were up to 29% slower than hand-coded message passing though they were within 9% of the performance of message-passing codes generated by XHPF [9]. A study of regular applications on SDSM by Han, Tseng and Keleher [12] touted improvements due to three compiler-directed strategies but their reported speedups in some cases significantly lag those reported by others for compiler-generated message passing implementations.*

Previous studies of regular applications on SDSM use one-dimensional data distributions and computation partitionings exclusively. Exorbitant communication would result due to the false sharing and fragmentation across distributed data dimensions other than the slowest varying one, if multi-dimensional data distribution and computation partitioning are used. However, applications with directional sweeps across different data dimensions, such as those common in computational fluid dynamics codes (e.g. [6]), can benefit from multi-dimensional partitioning to avoid over-sequentializing computation. Moreover, multi-dimensional partitioning is more scalable than one-dimensional partitioning because it increases the volume/surface ratio of the distributed array blocks so as to reduce the communication overhead. Optimizations to existent SDSMs to combat false sharing and fragmentation are crucial in making it feasible to use multi-dimensional data distributions and computation partitionings on SDSM.

Previous studies of regular applications on SDSM have focused on codes with loosely-synchronous parallelism, namely codes with completely parallel loop nests that are separated by synchronization [24, 9, 12, 8]. Noticeably absent in these studies are regular applications that require more tightly-coupled synchronization, such as that occurring in wavefront parallelizations of applications with loop-carried data dependences. If SDSM systems are to serve as a practical communication layer for targeting parallelizing compilers regardless of application style, it is important that they achieve good performance for a broader range of application styles. To date, SDSM systems have focused on barrier-style and lock-based synchronization, with

*For example, the speedup reported by Han, Keleher and Tseng [12] for a parallelized version of the SPEC benchmark TOMCATV using their CVS SDSM on an IBM SP2 was 3.6 for a 514x514 problem size on 16 processors whereas Adve and Mellor-Crummey reported a speedup of 11.36 for the same problem size on an IBM SP2 for message-passing code generated by the Rice dHPF compiler [3].

some special attention devoted to reductions. For tightly-coupled parallel applications, these synchronization primitives are less than ideal. Neither locks or barriers are well-suited for supporting wavefront parallelism, even though it is possible to use them for this purpose with careful hand coding.

1.2 Thesis Statement and Contributions

In this thesis, we study the performance issues of SDSM and develop integrated compiler and runtime SDSM strategies that target these issues. We focus on compiler and run-time support for improving the performance of regular applications on SDSM. Some of the compiler and run-time mechanisms we describe also apply to codes with irregular access patterns, although irregular applications are outside the scope of the current study.

The thesis of this work is that with proper compiler support and sophisticated runtime implementation, SDSM can be optimized to achieve good performance, so that it can provide a simple and efficient platform for both regular and irregular scientific parallel computation. To improve performance, we rely on compiler-derived knowledge of sharing and communication patterns to choreograph data movement and synchronization on a SDSM.

A distinguishing feature of our work is that we investigate techniques aimed at reducing the cost of false sharing and fragmentation caused by multi-dimensional data distributions and computation partitionings. These two strategies greatly increase the flexibility and effectiveness of parallelization of regular applications on SDSM. Our techniques for this purpose include compiler-restricted consistency, which reduces the cost of false sharing, and compiler-managed communication buffers, which reduces the cost of fragmentation when combined with compiler-restricted consistency.

Another distinguishing feature of our work is that we focus on regular applications characterized by wavefront computation and tightly-coupled sharing due to carried data dependences, since we found a significant lack of previous SDSM studies of these application types. We describe how we use point-to-point synchronization to effectively and efficiently parallelize this type of applications on SDSM.

By incorporating our novel techniques along with the optimization of compiler-controlled eager update into our point-to-point synchronization framework, we are able to efficiently and effectively support not only loosely-synchronous parallelism with multi-dimensional data partitioning but also tightly-coupled parallelism as well. On the 16-processor run, we achieve a speedup of 13 for a SOR benchmark of a 8Kx8K problem – a tightly-coupled computation based on wavefront parallelism †, and a speedup of 10 for the NAS-BT Class A application benchmark, which uses (block, block) ‡ 2D partitioning and applies wavefront parallelism across different data dimensions.

In addition, we demonstrate that these techniques even improve performance for a loosely-synchronous Jacobi benchmark for large problem sizes, when using multi-dimensional partitioning and point-to-point synchronization. For instance, we improve the speed up by 10% for Jacobi benchmark of problem size 4Kx4K and halve the amount of communicated data when using a (block, block) partitioning and point-to-point synchronization.

As the platform for this work, we use the dHPF compiler [3, 2, 23] and the TreadMarks SDSM system [4], both developed at Rice University.

1.3 Outline of Thesis

In the next chapter, we describe TreadMarks and the dHPF Compiler, the software platforms that serve as the basis of our investigations. In Chapter 3, we describe several key challenges for achieving high performance for large-scale regular applications

†The SOR benchmark here is different from the Red Black SOR benchmark that has been widely studied. In Red black SOR, new value is computed from only old values, while in SOR, new value is computed from both old values and new values.

‡Computation is divided up so each processor works on one of 16 square tiles.

on SDSM systems including synchronization efficiency, operation latency, data movement, and memory consistency. We describe our compiler techniques and new SDSM mechanisms that address these challenges. We explain how we exploit compiler-based knowledge of sharing and communication patterns to reduce false sharing by restricting the scope of memory consistency operations performed by an SDSM, reduce communication latency, and reduce communication cost by using bulk-data transfer. We show how our compiler supports efficient parallelization of tightly-coupled computation using our point-to-point synchronization framework, and how our compiler transforms "off processor" references to use shared buffers to reduce the cost of fragmentation when used together with compiler-restricted consistency. In Chapter 4, we present an experimental evaluation of the impact of our integrated compiler and SDSM mechanisms. In Chapter 5, we compare our work with related work, and discuss the pros and cons of different approaches. In the final chapter, we critically evaluate the approaches we propose, describe ways in which performance could be improved, describe unfinished work and evaluate the long-term prospects for SDSM for tightly-coupled large-scale computations based on regular access patterns.

Chapter 2

Background

In this chapter, we introduce the software components that serve as the foundation for our research.

2.1 TreadMarks Overview

As the starting point of our work, we use TreadMarks, an efficient page-based SDSM system from Rice University [4, 19]. It runs on commonly available Unix systems. DSM supports a shared memory programming model that provides a programmer with a single large address space. Data elements can be accessed within that address space much as they would be on a single machine. Software distributed shared memory provides a simple programming model for clusters of workstations that lack the special hardware support for consistency found in hardware multiprocessor machines. TreadMarks uses virtual memory protection hardware and a segmentation fault handler to maintain the consistency of the global shared address space and transfer data on demand between processors. Below, we describe TreadMarks in more detail.

Early SDSM Systems such as IVY [22], use the sequential memory consistency model (SC). In this model, a page can have only one writable copy and/or one or more read-only copies. Any write operation to a page by any processor will cause all the remote copies of the page residing on other processors to be invalidated or updated. Therefore, the amount of communication involved in SC is very high. In addition, false sharing caused by multiple writers to the same page, causes "ping-pong-ing", repeated transfers of a page between processors.

While early SDSM Systems used SC, modern SDSM systems typically use a more relaxed memory consistency model, the release consistency (RC) [18]. Shared memory accesses are usually protected by synchronization operations, which are divided into *acquire* and *release* operations. Acquires and releases roughly correspond to synchronization operations on a lock, but other synchronization mechanisms can be implemented on top of this model as well. For instance, arrival at a barrier can be modeled as a release, and departure from a barrier as an acquire. RC avoids much of the communication overhead that could otherwise occur if a SC model is used. In SC, each shared memory write access will cause communication to update any cached copies of the value on other processors. In contrast, RC only requires that before a processor may continue past a synchronizing acquire, write accesses that precede the synchronization be reflected at the acquiring processor. In other words, the shared memory is only consistent on release-acquire edges when using RC instead of SC. Consistency information is propagated along synchronization chains. Multiple write accesses may be buffered until a synchronization point is reached, and only the acquiring processor becomes aware of changes after the synchronization. To ensure that changes to shared data are visible, explicit synchronization must be present. This condition is already satisfied by concurrent programs that contain sufficient synchronization to avoid data races and nondeterministic results. Most concurrent programs fall into this class.

TreadMarks implements a lazy invalidate version of RC [18, 1]. It delays the propagation of consistency information until the time of an acquire. At an acquire, the releaser notifies the acquirer which pages have been modified, causing the acquirer to invalidate its copy of the modified pages. The acquiring processor later incurs a protection violation upon the first access to an invalidated page. At this time, it initiates communication to the processors that have most recently modified the page, asking for updates. TreadMarks implements a lazy invalidate version of RC because without compiler support, lazy invalidation has the lowest communication cost among the protocols tested [17].

TreadMarks uses a multiple-writer protocol [16] to reduce the cost of false sharing. In a software distributed shared memory system, memory consistency is maintained at the large granularity of pages. If a single-writer protocol is used, where only one writable copy of the page can exist at any time and only one processor can change the page, expensive message traffic will result due to the mutual invalidation or update of the page by multiple processors. This traffic is known as the ping pong effect. With a multiple writer protocol, two or more processors can simultaneously and independently modify their own copy of a shared page. Their modifications can be merged at synchronization point, thereby reducing cost of communication.

To reduce the amount of data transfered between processors, TreadMarks employs a *twin* and *diff* mechanism. Before the first modification to a validate page, a processor saves a copy of the page, called a twin. When other processors request updates made by this processor, it sends out only a diff, generated by comparing the page to its twin. This reduces the amount of data transfer.

The execution of each process is divided into a sequence of intervals, each denoted by an interval index. Every time a processor executes a release or acquire, a new interval is created, and the interval index is incremented. Intervals of different processes are partially ordered by synchronization. The partial order can be concisely represented by assigning a vector time stamp to each interval. The entry for processor p in the vector timestamp of interval i of processor p is equal to i. The entry for processor $q \neq p$ denotes the most recent interval of processor q that precedes the current interval of processor p. A processor computes a new vector timestamp at an acquire as the maximum of its previous vector timestamp and the releaser's vector timestamp.

Before a processor p may continue past an acquire from q, updates from all intervals with a smaller vector timestamp than q's current vector timestamp must be visible at p. Therefore, at an acquire, p sends its current vector timestamp to the previous releaser, q. On the release-acquire message to p, Processor q then piggybacks write notices for all intervals that p needs to know to maintain the release consistency. A write notice is an indication that a page has been modified in a particular interval, but it does not contain the actual modifications. The arrival of a write notice causes the corresponding page to be invalidated.

TreadMarks postpones diff creation until a processor requests the modification to a page, or a write notice arrives for a dirty page. In the latter case, it is essential to make a diff in order to distinguish the modification made by the different processors.

Access to an invalid page causes an access miss, which is handled by the SIGSEGV handler set up by TreadMarks to retrieve and apply to the page all the diffs that were created during the intervals that precede in the partial order the interval in which the page fault occurs. The faulting processor examines this page for those write notices, for which the processor does not have the corresponding diffs, and then requests the diffs in parallel. When all necessary diffs have been received, they are applied in increasing vector timestamp order.

The TreadMarks API contains a small number of functions. The synchronization primitives provided are locks, condition variables and barriers. `Tmk_lock_release` and `Tmk_lock_acquire` are calls to release and acquire the exclusive locks. `Tmk_lock_cond_signal`, `Tmk_lock_cond_wait`, and `Tmk_lock_cond_broadcast` provides synchronization for monitor semantics. Another synchronization operation, `Tmk_barrier`, permits system-wide synchronization and also allows garbage collection of data maintained for managing consistency such as write notices and diffs.

2.2 dHPF Compiler Overview

The Rice dHPF compiler is a research prototype compiler for High Performance Fortran [3, 2, 23]. The compiler translates HPF programs into Fortran 77 programs that use message passing calls for interprocess communication.

The dHPF compiler targets principally HPF with extensions that would broaden HPF's applicability or enhance performance. The compiler currently supports block

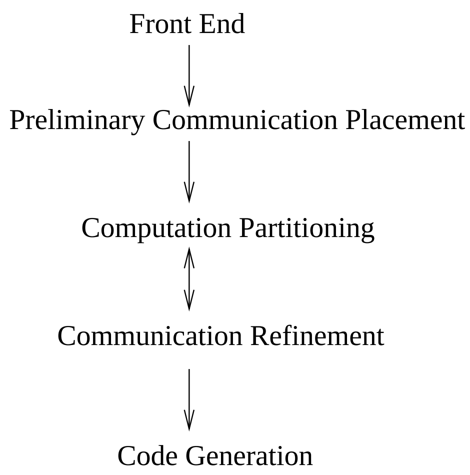

Figure 2.1 : dHPF Compiler Organization

and cyclic HPF data mapping directives for regular problems. The dHPF compiler features advanced computation partitioning, communication optimization and code-generation techniques. The compiler incorporates support for a flexible *non-owner-computes* computation partitioning model along with a variety of sophisticated code transformations to improve parallel performance including support for coarse-grain pipelining of wavefront computations and specialized handling of reductions. It provides an extensible platform for experimental research on compilation techniques and programming tools.

The organization of dHPF compiler is depicted below as in Figure 2.1. The front end is responsible for parsing, semantic checking of HPF directives, and preprocessing code for further analysis. All non-distributed variables are interpreted as replicated. Intraprocedural flow-sensitive propagation of (RE)ALIGN, (RE)DISTRIBUTE to statements and array references is done at this compilation stage. In addition, dHPF compiler front end also scalarizes F90 array section operations into F77 loops.

The preliminary communication placement module provides conservative information to the computation partitioner about where communication might be needed.

For each non-replicated variable reference, it determines where communication might be needed and hoists it to the outermost loop level possible while respecting data dependences on the reference and its subscripts.

The computation partition (CP) selector evaluates and selects from several computation partitioning alternatives that are not restricted only to the owner-computes rule. Starting from the innermost loop, it explicitly enumerates candidate partitioning choices for one loop at a time. Individual statements in a loop can have different computation partitions. The communication refinement module takes the computation partition choices computed by the CP selector and determines where the communication would be required. A cost function evaluates the cost of the potential communication. Cost estimates are based on communication frequency and communication volume. The CP selector uses the cost information to select the CP with the lowest communication cost.

The code generator generates SPMD F77 program from the CP and communication information computed at earlier compilation stages. It uses the Omega Library [20] developed at the University of Maryland as a building block for generating a sequence of SPMD loop nest sections and loop nests to pack or unpack message buffers. The dHPF code generator operates in three steps. First, it partitions the computation by reducing loop bounds and inserting guards where necessary. In this step, it also separates loop iterations that might access non-local values to minimize overhead from runtime locality checks. Next, it computes data sets to send/recv between processor pairs and generate code to pack/unpack buffers and send/recv data. Finally, it inserts storage management code. The storage management code can support REALIGN/REDISTRIBUTE and dynamic layouts.

Chapter 3

Compiler and Runtime Support

In this chapter we outline the performance challenges of using SDSM as a communication layer for parallel computation and describe extensions to the TreadMarks SDSM and to the dHPF compiler to support compiler and runtime cooperation for improving the performance of regular applications.

3.1 Challenges

There are three major challenges that must be met if codes using SDSM are to achieve performance comparable to codes using direct message passing for a wide variety of regular scientific applications.

Efficient Synchronization. In message-passing systems, interprocess synchronization is a side-effect of data communication. On hardware shared memory systems synchronization can be done at the cost of a few memory references. In contrast, on a SDSM an operation on a non-local synchronization object requires at least a round trip exchange of messages, incurring network delays in addition to message and interrupt handling at the endpoints. For performance, it is important to use a minimal number of synchronization operations, to overlap synchronization with computation, and to try to use each message for more than one purpose. While synchronization for ordering computation can be simulated using shared variables and locks, and reductions can be implemented using shared variables and multiple barriers, these approaches require more than the minimal number of operations. Direct implementations are required for efficiency. While barriers induce a global ordering on computations, they do not map easily onto pipelined computations, nor do they facilitate wavefront parallelism in which the shutdown of one pipeline can be overlapped with the startup phase of the next.

Operation Latency. In addition to round trip latencies incurred for synchronization operations, SDSMs that use invalidation-based memory consistency protocols (*e.g.*, Treadmarks [17]) also incur a round trip latency every time a logical block of data is copied between processors. Because data transfers occur on demand, when a processor faults on a memory location it incurs a round trip latency that cannot be overlapped with computation. For efficiency, round trip delays should be avoided by replacing them with single messages overlapped with computation, and by aggregating multiple operations into larger ones.

Excess Data Movement. There are several sources of excess data movement in SDSM systems. With the exception of a few hardware-dependent systems (e.g., [28]) and object-based systems [15], most SDSMs use software pages as the memory consistency block size. Software page size may be a multiple of hardware page size to help reduce the number of consistency operations needed to transfer a given volume of data. Page-size blocks cause two problems: false sharing and fragmentation. Both cause excess communication and consistency operations at synchronization points. With false shared pages, communication and consistency operations at synchronization points are useless. In the presence of fragmentation, whole blocks are communicated and made consistent when only a small fraction of the data is actually shared.

Some researchers have experimented with eager-update consistency protocols as a means of eliminating round trip latencies and overlapping communication with computation. A weakness of this strategy is that too much data can be sent without a precise analysis of sharing behavior [10].

3.2 Approach

We use an integrated approach involving both compiler and runtime support to address the challenges described in the previous section. The principal novelty of our approach is the introduction of compiler-managed *restricted consistency* to eliminate or delay communication caused by false sharing. We also provide compiler support for orchestrating data movement that leverages restricted consistency to virtually eliminate fragmentation. These mechanisms, in conjunction with support for point-to-point synchronization and selective eager update, dramatically reduce excess communication.

Compiler-Managed Restricted Consistency. Whenever processors synchronize, the multiple-writers protocol used in TreadMarks will re-establish the consistency of falsely-shared pages, even if they will continue to be falsely-shared. When there is a large number of such pages, this can be a large and unnecessary cost. To avoid such unnecessary consistency maintenance, we weaken the standard lazy release consistency model [4] by having a `signal` operation create consistency meta-data *only* for modified pages that *might* be accessed by its synchronization partner.

In general, the compiler proves that some set of data pages are guaranteed not to be shared between this synchronization operation and the next, and enforces consistency only for pages outside this set. For irregular applications, our analysis is inexact but conservative. However, for regular applications, the dHPF compiler computes precisely the set of pages that must be communicated. To eliminate round-trip delays, this set of pages can be updated eagerly.

Compiler-Managed Communication Buffers. As noted earlier, multi-dimensional computation partitionings can cause substantial fragmentation and false sharing. When pages contain only a few shared values, the dHPF compiler marshals the actively shared data into separate out-of-band, "communication" buffers. Moving the actively shared data to a compact set of densely packed pages enables restricted consistency to skip consistency operations on the fragmented pages.

Synchronization Mechanisms. For programs with a statically-defined mapping of computation to processors and regular data access patterns, data is shared pairwise between processors and synchronization is used to order computation to enforce dataflow constraints. To coordinate such sharing efficiently on the TreadMarks SDSM, we extended the application programming interface with support for point-to-point synchronization primitives `signal` and `wait`. We also extended the barrier mechanism to carry data on the barrier messages. This has many uses, but in our case the motivation is the efficient implementation of *reductions*.

Selective Eager Update. To eliminate round-trip communication for each pairwise-shared page following synchronization operations, we augmented our TreadMarks' implementation of point-to-point synchronization to support *selective eager update*. When a signal operation is performed, the update mechanism pushes specified pages of application data to the synchronization partner along with consistency metadata in anticipation of the waiter's expected future requests. The compiler is conservative, so this set of pages will be a subset of the data accessed after the synchronization. The number of pages sent is restricted to avoid flow-control and buffer management overhead associated with large messages. For regular applications, our compiler provides the exact set of pages to be communicated, and the SDSM runtime decides how many pages can be pushed without overwhelming the communication subsystem.

As our experiments in Section 4 show, the four mechanisms discussed in this section are complementary and each contributes in a significant way to overall application performance.

3.3 SDSM Extensions

We describe our protocol extensions to the API of the Treadmarks SDSM followed by a description of their implementation.

3.3.1 Point-to-point Synchronization

Given a statically-defined mapping of computation to processors, for regular data access patterns, parallelizing compilers can often determine (symbolically) at compile time exactly which processors need to synchronize. To efficiently support pairwise sharing found in many applications, we added point-to-point synchronization primitives `signal` and `wait` to the Treadmarks API for use by parallelizing compilers.

Pairwise synchronization has many advantages over barrier synchronization. Typically, pairwise synchronization is more efficient when it can be determined at compile time which processors need to synchronize. Barriers may over-constrain the program and reduce the amount of parallelism. In addition, a performance bottleneck typical in SDSM barrier implementation, due to the use of a centralized barrier manager, makes barrier synchronization expensive and not scalable. For applications that require only coordination between pairs of processors or subgroups of processors, pairwise synchronization is the best alternative. For example, pairwise synchronization is very convenient to implement coarse grain pipelining for loops that have wavefront parallelism.

It is also much more efficient and easier to selectively propagate some updates eagerly using pairwise synchronization primitives than a barrier synchronization primitive. With pairwise synchronization, the release message inherently goes in only one direction, from the releaser to the acquirer. Sending data along with synchronization messages is straightforward. Barrier synchronization involves bidirectional release messages, and each barrier call has to take a page list input for every processor. Implementing an eager update protocol based on barriers potentially increases the amount of messages from $O(p)$ to $O(p^2)$, where p is the number of processors.

The API of our pairwise synchronization primitives are:

`signal`(**int** acquirerId)

`wait`(**int** releaserId)

A releaser calls `signal` to notify the completion of events that the next operations of the acquirer depend upon. An acquirer calls `wait` to block until the corresponding notification arrives.

There are a number of ways of implementing pairwise synchronization primitives. One is a monitor-based approach that uses the locks and condition variables API provided by TreadMarks. In this approach, p^2 shared event counters would be allocated in the TreadMarks shared memory, where p is the number of processors executing the parallel application. The shared event counters would be tested and set when the synchronization primitives are invoked. Each event counter would use a distinct lock to ensure exclusive accesses. Since TreadMarks is implemented with a release consistency protocol, locks are necessary for processors executing the synchronization primitives to have a consistent view of the shared event counters. Condition calls would be used to block and resume program executions properly. This approach builds pairwise synchronization on top of available TreadMarks API, so it is easy to implement. However, the performance would not be very good because there are extra communications caused by maintaining a consistent view of the event counters and more synchronization than necessary.

Indeed, we chose to implement pairwise synchronization directly in TreadMarks. Every node maintains a vector of p event counters in the local memory of the releaser, where p is the number of processors. Associated with each event counter is a registry for the incoming synchronization acquiring request.

When the releaser executes a `signal` call, it first creates a new interval, and then builds consistency meta-data (called write notices) for all the dirty pages. Next, it updates and tests the event counter to see if the matching acquirer has sent its request. If so, the acquire will be blocked in the corresponding `wait` call; the releaser replies with a grant message and piggybacks the write notices. If not, the releaser simply returns.

When the acquirer executes a `wait` call, it sends out a request message along with its vector timestamp to the corresponding releaser asking for write notices and permission to proceed. The request message will cause a SIGIO signal, which traps the releaser into a SIGIO handler for this request. The SIGIO handler will update and test the event counter to see if the releaser has already executed the corresponding `signal` call. If so, it replies to the requester with a grant message piggybacked with write notices. Otherwise, the SIGIO handler simply returns.

Figure 3.1 shows pseudo-code for point-to-point `signal` and `wait` operations. Also shown are code for wait request handler that each node executes asynchronously for incoming messages from waiters.

3.3.2 Controlled Eager Update

To improve the efficiency of point-to-point synchronization, the `signal` primitive has been enhanced to support *controlled eager update*. When eager update is enabled, a `signal` operation sends application data for the specified set of pages to the synchronization partner along with the usual consistency meta-data. The partner uses this to eagerly update copies of its application data. When the data is referenced by the partner after it executes its matching `wait`, the data will be consistent so neither page faults nor communication is necessary at that time. This protocol can reduce overhead of software DSM runtime and communication. Bulk data transfer and piggybacking data with synchronization release messages can be incorporated with this protocol seamlessly.

If eager update is enabled, page diffs are computed for the set of application data pages specified as an argument to the `signal` operation. These diffs are then packed together with consistency meta-data in large messages, and sent to the synchroniza-

```
procedure signal(int acquirerId)
  block interrupts;
  compute consistency meta-data for all dirty pages;
  if(matching wait_request received)
     reply with consistency meta-data;
  unblock interrupts;
}

procedure wait(int releaserId){
  block interrupts;
  while(releaser has not replied)
    send request to the releaser and sit receiving;
  unblock interrupts;
}

void wait_request_sigio_handler(Acquire *acq){
  register request into the pre-allocated buffer;
  if(corresponding release has been executed)
    reply with consistency meta-data;
  else return;
}
```

Figure 3.1 : Unoptimized implementation of point-to-point synchronization primitives.

tion partner. The data set that the compiler provides for eager update must be conservative in the sense that it is a subset of the exact communication data set. In other words, an inexact compiler analysis may be used to specify the data set, as long as we do not push extra data outside the exact communication set. Although pushing the wrong data does not affect the correctness of the program execution, it introduces unwanted communication.

The signaler maintains vector time stamps for data that it eagerly ships so that same diffs are not eagerly sent twice. These time vector records are maximized with the time vector records of incoming page diffs at a diff apply operation. The time vector records are conservatively incremented after the eager push operation, even though the messages sending the diffs may not be actually received. This is to avoid sending same diffs twice during the next eager update operation. However, this also means that all application data is not always received via eager push operation: some is received on demand in a lazy request fashion.

At the waiter, a SIGIO handler receives and saves the eagerly pushed messages. Processing of these messages is postponed by the receiver until it executes the corresponding `wait` operation. The advantage of delaying processing the message is to improve locality by avoiding the pollution of the receiver cache while it is in the middle of a computation.

When a `wait` is executed, an event counter is inspected to determine whether the synchronization partner has already provided consistency metadata and application data diffs eagerly. If not, a message is sent to the signaling partner to request the information. In any case, once the waiter has the necessary data, diffs are applied and pages are validated once all of the pending diffs have been received.

When the releaser must transfer a significant amount of data, update messages may arrive late. Thus, to prevent the acquirer to request prematurely, the acquirer is put to sleep and waits to be awakened by the arrival of messages or timeout.

A networking protocol that incorporates flow control is needed to throttle back the eager update protocol so that a signaler does not push too many messages at a time. A networking protocol that guarantees the delivery of messages is preferred but not required. If an unreliable protocol is used, `signal` messages may be lost. The receiver will need to request explicitly at least the consistency meta-data packet, if it has not received it after a time out event. On IBM SP2 systems, we implemented our controlled eager update protocol on top of IBM's MPL [29], a reliable networking protocol with flow control. The releaser copies the messages into a communication buffer in the user space, calls non-blocking send and returns. The send has to be non-blocking to avoid deadlock, because otherwise corresponding receives might not have a chance to execute to consume the messages that are being pushed. The communication buffer space is freed and recycled when the signaler later invokes a wait operation.

3.3.3 Restricted Consistency

When two or more processors are false-sharing a page, lazy release consistency can induce a firestorm of consistency traffic to unnecessarily update these falsely shared pages at synchronization points. To avoid such unnecessary consistency maintenance, we weaken the standard lazy release consistency model [4] by having a `signal` operation build consistency meta-data only for the specified set of pages that might be subject to true sharing by the synchronization partner.

In this consistency model, compilers can prove that some pages do not need to be communicated among processors, so write notices for these pages do not need to be built when an interval is created. Thus, after synchronization, one processor may not need to see the changes of these pages by another processor as long as the compiler can prove it is safe.

There are three data sets that the compiler may provide to the SDSM `signal` operation for restricted consistency. First, the set can be the actual communication data set that the partner processor will reference nonlocally after the `wait` operation before the next `wait` operation. We name it *ExactComm* set. Second, in the case where the compiler can not determine exactly what the ExactComm set is, it can provide a conservative estimate set, which is a superset of *ExactComm* set. Here, we name it *EstimateComm* set, since it is the data set that the partner processor might touch after the `wait` operation. Third, the compiler can give a *NoComm* set, which the partner processor will not touch after the `wait` operation until after the next `wait` operation. The *Estimate* set is the complement of the *NoComm* set. The *ExactComm* set is the lower bound set of the *EstimateComm* set. The closer the *EstimateComm* set is to the *ExactComm* set, the more effective the compiler-restricted consistency is.

For regular applications, since the compiler can easily determine the *ExactComm* set, the data set for which consistency must be enforced is exactly the same set as the data set for controlled eager update. Therefore, when using the restricted consistency protocol, we only compute consistency meta-data only for dirty pages that are specified in the set of pages provided as an argument to the `signal` operation. Other dirty pages are simply left marked as dirty.

Therefore, at the interval creation we intersect the page range list provided by the compiler with the dirty page range list, and build write notices only for those pages that are present in both page range lists. The dirty page ranges are split and non-intersecting parts are put back, if only part of the range is in the other list. To efficiently do the intersection and splitting, the page range list provided by the compiler is sorted when it is built. Those dirty pages proved to be falsely shared are untouched in the current round of synchronization, just as if the writes to these pages are accumulated and postponed until a later synchronization where they become truly shared in some sense.

To efficiently recycle memory, we modified the TreadMarks garbage collecting mechanism and introduce *restricted consistency barrier* just for the purpose of garbage collection. During the restricted consistency barrier, no new intervals or write notices are created for dirty pages. After merging the existed consistency meta-data, old consistency data, diffs and twins should be freed. However, for those dirty pages that have not created a write notice due to the compiler controlled consistency, we need to keep their twins and dirty states for later interval creation. We still use the original full consistency barrier for global synchronization.

The restricted consistency implementation can also be orchestrated to support the other two types of data set. This is especially useful to support irregular applications. What is needed is minor changes to support barrier synchronization. The three data sets should extend to be the union data set that all other processors might touch after barrier synchronization instead of just a partner processor. For example, the *EstimateComm* set means the data set that might be referenced nonlocally by any other processor after the barrier synchronization. The implementation of creation of intervals and write notices remains the same as our implementation for regular applications.

Figure 3.2 is an irregular example code representative of many real applications using finite element method on unstructured meshes. For example, one such application is `Fem-3D` that solves the equilibrium equations in three dimensions. Specifically, it evaluates the elemental stiffness matrices, computes the displacements, and stresses at the quadrature points using the Conjugate Gradient method on unassembled stiffness matrices [13, 14]. In Figure 3.2, array A and array B are initialized before entering a loop of computation. The reference to array A causes communication because it uses value computed from last iteration on different processors. The reference to array B should not cause any communication except in the entry iteration where it needs the value initialized on different processors. Without restricted consistency, the original code will cause communication for false-shared pages in array B, when the partitioning of array B is not page aligned. However, with restricted consistency and loop peeling, the false sharing cost can be significantly reduced. To do this, we include both array A and array B in the *EstimateComm* set in *compute*1, while in

```
subroutine fem(nb)
  real A(0:N-1), B(0:N-1)
  common /tmk_shared/ A, B
  integer nb(0:N-1, 0:2) // a fixed neighbor list.

  call compute_neighbor(nb) // compute the neighbor list.

  call initialize() // initialize value to array A and array B.

  do i = 1, 1000
    call compute(nb)
  enddo
end  // of fem.

subroutine compute(nb)
  real A(0:N-1), B(0:N-1)
  common /tmk_shared/ A, B
  integer nb(0:N-1, 0:2)

  call tmk_barrier(Set_of_All) // consistent for all shared address space.

  do j = proc_id*N/nproc, proc_id*N/nproc + N/nproc
      B(j) = 0.3*(A(nb(j,0)) + A(nb(j,1)) + A(nb(j,2))) + 0.1*B(j)
  enddo

  call tmk_barrier(Set_of_All) // consistent for all shared address space.

  do j = proc_id*N/nproc, proc_id*N/nproc + N/nproc
      A[j] = B[j]
  enddo
end  // of compute.
```

Figure 3.2 : An irregular example that can benefit from restricted consistency

```
subroutine fem_peel(nb)
  real A(0:N-1), B(0:N-1)
  common /tmk_shared/ A, B
  integer nb(0:N-1, 0:2) // a fixed neighbor list.

  call compute_neighbor(nb) // compute the neighbor list.
  call initialize() // initialize value to array A and array B.
  call compute1(nb)
  do i = 2, 1000
    call compute2(nb)
  enddo
end  // of fem.

subroutine compute1(nb)
  real A(0:N-1), B(0:N-1)
  common /tmk_shared/ A, B
  integer nb(0:N-1, 0:2)

  call tmk_barrier(Set_A_and_B) // consistent for both array A and array B.
  do j = proc_id*N/nproc, proc_id*N/nproc + N/nproc
      B(j) = 0.3*(A(nb(j,0)) + A(nb(j,1)) + A(nb(j,2))) + 0.1*B(j)
  enddo
  call tmk_barrier()
  do j = proc_id*N/nproc, proc_id*N/nproc + N/nproc
      A[j] = B[j]
  enddo
end  // of compute1.

subroutine compute2(nb)
  real A(0:N-1), B(0:N-1)
  common /tmk_shared/ A, B
  integer nb(0:N-1, 0:2)

  call tmk_barrier(Set_A_Only) // consistent for array A.
  do j = proc_id*N/nproc, proc_id*N/nproc + N/nproc
      B(j) = 0.3*(A(nb(j,0)) + A(nb(j,1)) + A(nb(j,2))) + 0.1*B(j)
  enddo
  call tmk_barrier()
  do j = proc_id*N/nproc, proc_id*N/nproc + N/nproc
      A[j] = B[j]
  enddo
end  // of compute2.
```

Figure 3.3 : An irregular example using restricted consistency

*compute*2, we include only array A in the *EstimateComm* set. Figure 3.3 shows that we avoid false sharing in array B by providing restricted-consistency information to the *barrier* calls.

3.3.4 Put All Together

Figure 3.4 shows pseudo-code to support `controlled eager update` and `restricted consistency` on our point-to-point synchronization primitives. The interface for `signal` accepts a set of pages (specified as a list of page ranges) that are used both for `controlled eager update` and `restricted-consistency`. Also shown are code for handlers for wait request and update messages that execute in response to SIGIO interrupts.

To run large benchmarks such as NAS-BT, we optimized the memory usage of TreadMarks. TreadMarks creates twins to distinguish the the old copy of the page from the new copy of the page. However, since the large inner section of the array does not need to be communicated or even diffed, we introduce *virtual twin* mechanism, which does not allocate a twin page when a page is twined for the first time. If the page is essentially private, it is not going to consume any memory for twins. Due to the same reason, we also introduce *virtual vector time stamp*, which dynamically allocates vector time stamp for those pages that need to be eagerly pushed.

3.4 Compiler Support

Here we describe extending the dHPF compiler to target shared memory and compiler support for optimizing SDSM performance.

3.4.1 Extending dHPF to Shared Memory

The dHPF compiler statically partitions computation among a set of participating processors. For regular data access patterns, wherever synchronization is needed to preserve data dependences, the compiler statically computes (symbolically) which

```
procedure signal(int acquirerId, PageRange[]) //releaser
  block interrupts;
  Compute consistency meta-data for PageRange;
  if (matching wait_request received)
      Reply with consistency meta-data and diffs(application data);
  else
      if(eager-update enabled) // push
          Build  update_request;
            Select message content based on
                approximate timestamp
                and network properties;
          Transmit using non-blocking send;
  unblock interrupts;
  return;

procedure wait(int releaserId) //acquirer
  block interrupts;
  if(eager-update enabled)
      if(matching update_request has been received)
           // fall through
      else // wait for update before sending explicit request
          set timer and unblock interrupts;
          while(update_request not received && not time out)
              poll the sigio handler, to receive update_request;
          block interrupts;
  if(update_request unavailable) // i.e. eager-update not used or failed.
      send wait_request to releaserID and wait for reply;
  process consistency meta-data, apply diffs and validate pages;
  unblock_interrupts;
  return;

procedure wait_request_sigio_handler(Acquire *acq) // releaser
  register wait_request;
  if(corresponding signal has been executed)
      reply with consistency meta-data and diffs;
  return;

procedure update_request_sigio_handler(Update *updt) // acquirer
  register the incoming message;
  copy data to preallocated buffer;
  return;
```

Figure 3.4 : Optimized implementation of point-to-point synchronization primitives.

pairs of processors need to synchronize and generates calls to `signal` and `wait` operations for SDSM. All synchronization involves coordinating with a data owner (as determined from HPF distribution directives). Before a non-owning processor is allowed to access the data, it is granted access by a signal from the owner. When a processor has completed access to non-owned data, it signals the owner. The dHPF compiler automatically places signal/wait pairs for data dependences at the outermost loop level possible. If separate compatible pairwise synchronization operations for multiple arrays are placed at the same point in a program, dHPF collapses them into a single synchronization pair.

To extend the dHPF compiler code generator from message passing to target shared memory, we need to generate appropriate synchronization calls instead of the original communication calls. Our approach is not restricted to software distributed shared memory, it also includes any hardware shared memory as long as the runtime library supporting the synchronization primitives are provided. Shared memory programs use synchronizations to avoid race condition and ensure correct execution. For a software shared memory system like TreadMarks, which uses a release consistency protocol, synchronization calls are crucial to make memory changes viewable among different processing nodes.

In the following subsections, we explain the synchronization framework adopted by our dHPF compiler.

Dependence Analysis

In parallelizing a program that will use shared memory system for communication, synchronization is needed whenever there are a pair of conflicting memory references issued by different processors. A pair of memory references conflict if the two references are to the same memory location, and at least one of them is a memory write operation. Data dependence analysis is used to conservatively identify all possible conflicting memory references to provide information for the code generator to insert proper synchronization calls.

There are basically three important types of data dependences for avoiding conflicting concurrent accesses:

True Dependence: a `read` after a `write`

Anti Dependence: a `write` after a `read`

Output Dependence: a `write` after a `write`

The dHPF message passing compiler only considers true dependences and output dependences. Anti-dependences can be ignored when generating message passing communication code, because values are communicated using copies through message buffers. For instance, one processor can safely overwrite a value before a `read` operation by another processor, since the `read` operation will read the needed shared data from its own messages buffer, not from in-place data that may have been already changed by a concurrent `write`. It helps improve the degree of concurrency by not enforcing anti-dependences in the output SPMD code.

However, for a compiler targeting shared memory, all three types of data dependences need to be considered. Even in a software distributed shared memory system, where old versions of data are retained most of the time, we still need to enforce anti-dependences to ensure safe execution. For instance, TreadMarks uses lazy diff creation, synchronization has to be present to enforce partial ordering such as, a `read` operation happening before a `write` operation, otherwise overwritten value may be incorrectly encoded into the diffs and read.

Synchronization Strategy

To improve data locality, the HPF data parallel language requires data ownership specifications through ALIGN and DISTRIBUTE layout directives. In the shared-memory semantics, there is no such concept as ownership of shared data items. It is not essential to partition data explicitly even though computations are broken up among processors. Thus, shared memory provides more flexibility than message passing. However, it is often profitable to keep this notion of data ownership to improve locality, which has similar performance advantages as have been shown in home based SDSM systems [30]. For best performance, computations that access the same set of data should be assigned consistently to the same processor.

Our shared memory dHPF compiler uses data layout or ownership directives to classify references in the program into references that are always to owned "local" data and "nonlocal" references that sometimes access non-owned data. Conflicting memory operations may exist between a pair of local and nonlocal references or a pair of nonlocal references to the same memory location. No conflicts exist between any pair of local references to the same memory location, since local references are executed by the same processor and their dependence order is already maintained through sequential execution.

The message passing dHPF compiler generates a pair of send-receive calls for each nonlocal reference *. The communication calls are inserted at a point before the statement of nonlocal reference if the reference is a `read` operation. The communication calls are inserted after the statement of nonlocal reference if the reference is a `write` operation. For the shared memory, synchronization calls replaced communication calls. We use a *synchronizing through owner* rule. Two pairs of signal-wait calls are generated for each nonlocal reference unlike the message passing code where only one pair of communication calls are needed. One of them conservatively ensures potential anti-dependence or output dependence order. Before performing a nonlocal reference, a process must acquire permission from the owner. After finishing a nonlocal reference, a process must notify the owner that the operation has completed. Any two conflicting nonlocal references are synchronized indirectly through the owner process. The overhead with this indirect synchronization approach is small in practice since compiler uses the *owner computes* rule most of the time and dependences typically

*Actually, sends and receives for multiple nonlocal references are often coalesced to reduce communication volume and frequency.

occur only between a local reference and a nonlocal reference.

To reduce the communication cost, the message passing dHPF compiler places the communication calls at the outermost loop level that can still guarantee correct execution. This outermost loop level is called *communication level*. The communication level is bounded by the loop level of any true dependence and output dependence that is incident on the nonlocal reference. Similarly, for the shared memory dHPF compiler should place the synchronization calls at the outermost safe loop level, which is the *synchronization level*. Calculation of the synchronization level is more complicated. As described in the previous paragraph, there are two pairs of signal-wait calls for each nonlocal reference, one before the reference which we call *pre* synchronization and one after the reference which we call *post* synchronization. If the reference is a nonlocal `read`, then the `pre` synchronization is pinned by any true dependence that is incident upon the reference, which is the same as the placement of message passing calls, but the `post` synchronization is only pinned by any anti-dependence that is incident from the reference. Conversely, if the reference is a nonlocal `write`, then the `pre` synchronization is pinned by any anti-dependence and output dependence that is incident upon the reference and the `post` synchronization is pinned by any true and output dependence that is incident from the reference.

For regular applications, it is simple and efficient to use a pairwise synchronization scheme. Pairwise synchronization is principally implemented in terms of pairs of `signal` and `wait` runtime calls. An invocation of `wait` call will stall the process until the partner process executes the corresponding `signal` call. As stated before, a nonlocal reference is sandwiched between two pairs of synchronization calls. The `pre` synchronization calls are synchronization calls inserted before a nonlocal reference, and are called `signal_avail` and `wait_avail`. The `post` synchronization calls are synchronization calls inserted after a nonlocal reference, and are called `signal_done` and `wait_done`.

Figure 3.5 shows the synchronization strategy for regular data accesses. Prior

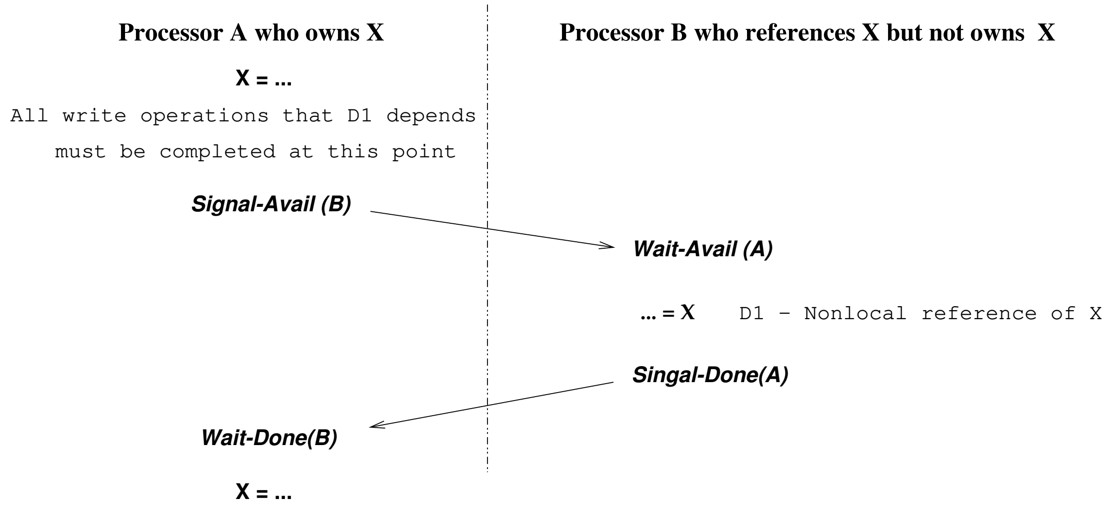

Figure 3.5 : Point-to-point synchronization illustration

to any nonlocal data access, the requester of the nonlocal access will execute a `wait_avail`, which will suspend the execution until the owner of the data item executes a corresponding `signal_avail`. The owner will execute the corresponding `signal_avail` if and only if all the data accesses that this nonlocal access depends on have already been completed. The owner also needs to execute a `wait_done` at some point after executing the `signal_avail` to wait for the nonlocal access to finish. The requester will execute a corresponding `signal_done` once it completes that nonlocal access. Therefore, all the data dependences are enforced through synchronizations with the owner.

Theoretically, we can also use pairwise synchronization for irregular applications. This can be done through runtime resolution, which involves computing the processor id to synchronize at runtime. However, this approach is inefficient. With runtime resolution, we are not able to reduce the loop bounds and the computation of the loop is essentially serial if this synchronization approach is adopted.

Therefore, barriers are introduced wherever there is a data dependence that involves irregular references. Nonlocal irregular references will be sandwiched between a pair of barrier calls, if it is involved in any data dependence. Like pairwise synchronization, barriers are hoisted to the outermost safe loop level without letting any data dependence be violated. To reduce communication cost, we can also move barriers of the same loop level with respect to data dependence and coalesce them together.

Special reduction barriers are used when dealing with reductions in our shared memory dHPF compiler.

Data Layout

To support generating shared memory Fortran code, we conservatively assume that all arrays with a partitioned HPF data distribution are shared, and we allocate them in shared common blocks. Thus, if the array is distributed and originally declared in a common block, say `foo`, we need to separate it out and allocate it in another shared common block, `foo_shared`. If the array is distributed but is not in any common block, we also need to allocate it in a shared common block, say `tmk_proc_name`, to be accessed by all the processors.

All shared memory is statically allocated by specifying the starting address of common blocks in a generated assembly code script passed to the loader.

3.4.2 Compiler Support of Eager Update and Restricted Consistency

As described in the last chapter, `restricted consistency` can effectively reduce the amount of false sharing, and `eager update` can replace excessive round-trip messages with fewer one-way large messages by direct signaling and data aggregation. However, without compiler support, none of them can fly. In this section, we explain our compiler extensions to provide useful analysis information to the SDSM runtime.

Both `restricted consistency` and `eager update` need compiler support, but their compiler analyses are different. Generally speaking, compiler should provide a subset of *ExactComm* set for `eager update` and a superset of *ExactComm* set for `restricted consistency`. However, for regular applications, where the compiler analysis is precise, the *ExactComm* set can be determined at compiler time. Therefore, we pass the same set for both `eager update` and `restricted consistency`.

For regular access patterns, dHPF computes regular sections describing the data that must be communicated at each synchronization point. At run-time, a regular section descriptor for an array that needs to be communicated, which is represented as a vector of (lower bound, upper bound, stride) tuples for each array data dimension, is passed to a run-time library routine along with the base address of the array and the layout description of the array. This routine produces a vector of page ranges that contain data described by the regular section. The page range vectors for all arrays that need to be communicated at a synchronization point are passed to the SDSM `signal` operation.

With the page range vectors, the SDSM runtime can eagerly push the data that is going to be referenced at the receiver side in big messages, so as to avoid round trip communication and aggregate data transfer. Furthermore, with the page range vectors, the SDSM runtime can avoid creating unnecessary meta data, so as to reduce false sharing.

In shared-memory code generator of dHPF compiler, we use rectangular section analysis to determine the data that is to be communicated. When generating synchronization calls, we pass a rectangular section descriptor to the TreadMarks runtime. A rectangular section descriptor is of form

$<$ SectionLowerBound0, SectionUpperBound0, Stride;
SectionLowerBound1, SectionUpperBound1, Stride;
... $>$

We also pass an array descriptor to TreadMarks runtime to calculate the correct pages that are going to be touched by the `acquirer` after synchronization sink. The array description is of a form

$<$ ArrayReference;
Type;
ArrayLowerBound0, ArrayUpperBound0;
ArrayLowerBound1, ArrayUpperBound1;
$\cdots$ $>$

The compiler and runtime support includes a number of functions to compute the descriptors. They are `dhpf_init_pagelist`, `dhpf_append_bound`, `dhpf_append_triple`, and `dhpf_append_pagelist`. The call to `dhpf_init_pagelist` initializes an empty page range list. The call to `dhpf_append_bound` passes the array descriptor to the runtime system. The call to `dhpf_append_triple` passes the rectangular descriptor to the runtime system. Lastly, the call to `dhpf_append_pagelist` computes and appends the page range list with new pages. To efficiently implement restricted consistency, new page ranges are merged and sorted in the call to `dhpf_append_pagelist`.

Synchronization calls inserted after a nonlocal read reference don't need to be passed in any descriptors, since the synchronization back from a nonlocal read is guaranteed not to cause any communication back to the owner. Other synchronization calls need to be fed with these compiler generated descriptors. This is obvious with both synchronization calls before a nonlocal read and synchronization calls after a nonlocal write, since the synchronization calls there are used to enforce true dependence and data communication is expected to happen. For synchronization calls inserted before a nonlocal write, data communication may occur because of output dependence. We still provide the descriptors speculatively to these synchronization calls before a nonlocal write, and rely on the TreadMarks runtime to avoid pushing outdated data.

3.4.3 Compiler Management of Explicit Communication Buffers.

For computations that involve multi-directional line sweeps through data, partitioning along only one dimension can overly sequentialize a line sweep along that dimension and reduce the efficiency of wavefront parallelism. However, fragmentation is a significant problem on SDSM when computing with multi-dimensional arrays using anything other than a one-dimensional data partitioning. For a rowwise computation partitioning of an array in standard Fortran column major layout in memory, adjacent processors will operate on copies of the same page. At synchronization points, consistency would be established for all of the data on these shared pages. It is typically the case that processors sharing the same page need not be informed of all of the values computed on that page by some adjacent processor. For stencil computations, just communicating the data values along the partitioning boundary would suffice.

To avoid unnecessarily communicating all of the data on such shared pages, the dHPF compiler supports copying data to be communicated into separate out-of-band page-aligned "communication" buffers. By writing only data that must be communicated into the buffer, the compiler ensures that resulting page diffs for the buffer are much smaller than the contents of all pages shared across a partitioning boundary. These buffers are filled with data that would be placed in overlap areas used by message-passing compilers [11].

The `explicit communication buffer` optimization has to be combined with `compiler-restricted consistency` to be really effective. The reason is that `explicit communication buffer` optimization only reduces fragmentation. It replaces nonlocal references that cause fragmentation with buffer references. However, other local references are usually located on the same pages as the original nonlocal references. Therefore, fragmentation is merely turned into false sharing plus extra communication via shared buffer. This usually results worse performance because of the increased communication. However, under the optimization of `restricted consistency`, false sharing due to local references of pages whose computation is partitioned across processor boundaries can be completely avoided, and there is only communication via the shared buffers.

The communication buffer is declared in the form of `name_direction_dimension(length, proc_dim1, proc_dim2, ...)`, where *name* is the same as the HPF array name, *direction* reflects whether the communication is from left to right, *dimension* means which array dimension the communication is crossing, *length* is the size of the buffer, *proc_dim1, ...* means which processor the buffer is allocated to. Before the synchronization, a loop generated by the compiler is used to pack the contents of the partition boundary into the buffer. The compiler also rewrites the later reference to the array partition boundary to directly index the value of the communication buffer. Note that the buffer is singular dimensional array for the purpose of aligning to be integral number of pages. Thus, the compiler has to linearize the reference indices in both packing and referencing the communication buffer. This requires the knowledge of the rectangular array section that the buffer will hold.

3.4.4 Example

In this section, we use examples to further illustrate our integrated compile-time run-time system. The HPF version of our example is shown Figure 3.6.

The iteration space of this program is derived from loops in NAS BT. Other applications like Sweep 3D and computational techniques such as Alternating Direction Integration [7] also have similar parallelism characteristics. For each time step, the first loop nest sweeps along the i dimension, making j loop parallel and i loop carried, while the second loop sweeps along the j dimension, making i loop parallel, and j loop carried.

To avoid false sharing, The barrier version uses a 1D partitioning, two copies of the array, and relies on an expensive transpose copy operation for communication. Otherwise, it can only parallelize one of the loops.

Therefore, we use 2D partitioning. Our dHPF TreadMarks system can parallelize both loops without expensive transpose. For each loop, there are two levels of parallelism, a parallel loop and a coarse grained pipelining. Our pairwise synchronization

```
      program example
        integer step, i, j, k
        parameter (n = 1024)
        real a(n, n)
CHPF$   processors p(4,4)
CHPF$   template t(n, n)
CHPF$   align a(i, j) with t(i, j)
CHPF$   distribute t(block,block) onto p

C       *** initialization ***
        do j = 1, n
           do i = 1, n
              a(i, j) = i * n + j
           enddo
        enddo
C       ***** iteration ******
        do step = 1, 100
           do j = 2, n - 1
              do i = 2, n - 1
                 a(i, j) = a(i, j) + 0.25*a(i - 1, j)
              enddo
           enddo
           do j = 2, n - 1
              do i = 2, n - 1
                 a(i, j) = a(i, j) + 0.25*a(i, j - 1)
              enddo
           enddo
        enddo
```

Figure 3.6 : The HPF code of an example program.

extensions to TreadMarks are critical in exploiting the pipelined parallelism. The optimized code dHPF compiler generated for the first loop is shown in Figure 3.7. The code for the second loop is similar as the first one, except it does not need to pack data into a shared buffer to reduce false sharing.

Line 45 through Line 52 is the buffer packing loop. Line 53 through Line 62 is the compiler supported runtime calls to create the rectangular section descriptors and array descriptors. Line 63 is the call to `signal_avail` with a page range list. Line 19 is the call to the corresponding `wait_avail`.

The synchronization calls to `signal_done` and `wait_done` at line 73 and line 80 are inserted conservatively to enforce anti-dependences. Since the anti-dependences are all carried at the time step loop level, these `post` synchronizations back to the owner is hoisted out to the loop level of the carried antidependence. The `pre` synchronizations `signal_avail` and `wait_avail` are pinned by true dependence carried on i loop. However, they appear at the j loop level, because i loop is iterating through the local section of a partitioned data dimension and synchronization happens at processor partition boundaries.

```
      program example
        dimension pages(1:32768), tuples(1:1024), bounds(1:1024)
        dimension a_lower_buffer_1(0:2047, 0:3, 0:3), a_lower_buffer_0(0
     *:2047, 0:3, 0:3), a(1:1024, 1:1024)
        common /tmk_shared_common_example/ a
        common /buffer_common_example/ a_lower_buffer_0, a_lower_buffer_
     *1

C       initialization loop omitted

         do j_dtmp__blk = 256 * p_myid2 + 1, 256 * p_myid2 + 249, 8

           p_sync1 = p_myid1 - 1
           p_sync2 = p_myid2
           if (p_myid1 .ge. 1 .and. 256 * p_myid2 .le. j_dtmp__blk + 6 .a
     *nd. j_dtmp__blk .le. 1023 .and. 256 * p_myid2 .ge. j_dtmp__blk - 2
     *56 .and. j_dtmp__blk .ge. (-5)) then
             partner_rank_0 = p_sync1 + p_sync2 * 4
             call dhpf_wait_avail(partner_rank_0, (myid))
           endif

           do j = max(j_dtmp__blk, 128 * p_myid2 + 2), min(192 * p_myid2
     *+ 447, j_dtmp__blk + 7)
             do i = max(256 * p_myid1 + 1, 2), min(256 * p_myid1 + 256, 1
     *023)
               if (p_myid1 * 256 .le. i - 2 .and. i - 2 .lt. p_myid1 * 25
     *6 + 256 .and. p_myid2 * 256 .le. j - 1 .and. j - 1 .lt. p_myid2 *
     *256 + 256) then
                 other_ref_0 = a(i - 1, j)
               else
                 other_ref_0 = a_lower_buffer_0((j - (p_myid2 * 256 + 1))
     * * 1 + (i - 1 - (p_myid1 * 256 - 1 + 1)), p_myid1, p_myid2)
               endif
               a(i, j) = 0.25 * other_ref_0
             enddo
           enddo

           p_sync1 = p_myid1 + 1
           p_sync2 = p_myid2
           if (j_dtmp__blk .le. 1023 .and. p_myid1 .le. 2 .and. 256 * p_m
     *yid2 .le. j_dtmp__blk + 6 .and. 256 * p_myid2 .ge. j_dtmp__blk - 2
     *56 .and. j_dtmp__blk .ge. (-5)) then
             partner_rank_0 = p_sync1 + p_sync2 * 4
             call dhpf_init_pagelist(pages, (myid))
C            --< Pack Buffer >--
             do D_i1 = max(256 * p_myid2 + 1, j_dtmp__blk, 2), min(256 *
     *p_myid2 + 256, j_dtmp__blk + 7, 1023)
               D_i2 = 256 * p_send_id1
               packreal = a(D_i2, D_i1)
               a_lower_buffer_0((D_i1 - (p_sync2 * 256 + 1)) * 1 + (D_i2
     *- (p_sync1 * 256 - 1 + 1)), p_sync1, p_sync2) = packreal
```

```
            enddo
            call dhpf_append_bound(bounds, 1, 0, 0, (myid))
            call dhpf_append_bound(bounds, 2, 0, 255, (myid))
            call dhpf_append_triple(tuples, 1, 256 * p_myid1 + 256 - (p_s
     *ync1 * 256 - 1 + 1), 256 * p_myid1 + 256 - (p_sync1 * 256 - 1 + 1)
     *, 1, (myid))
            call dhpf_append_triple(tuples, 2, max(j_dtmp__blk, 256 * p_m
     *yid2 + 1, 2) - (p_sync2 * 256 + 1), min(256 * p_myid2 + 256, j_dtm
     *p__blk + 7, 1023) - (p_sync2 * 256 + 1), 1, (myid))
            call dhpf_append_pagelist(pages, a_lower_buffer_0(0, p_sync1,
     * p_sync2), bounds, tuples, 2, 4, (myid))
            call dhpf_signal_avail((myid), partner_rank_0, pages)
          endif

        enddo

        p_send_id1 = p_myid1 - 1
        p_send_id2 = p_myid2
        if (p_myid1 .ge. 1) then
          partner_rank_0 = p_send_id1 + p_send_id2 * 4
          call dhpf_init_pagelist(pages, (myid))
          call dhpf_signal_done(partner_rank_0, (myid), pages)
        endif

        p_recv_id1 = p_myid1 + 1
        p_recv_id2 = p_myid2
        if (p_myid1 .le. 2) then
          partner_rank_0 = p_recv_id1 + p_recv_id2 * 4
          call dhpf_wait_done((myid), partner_rank_0)
        endif

         ...
```

Figure 3.7 : The F77 SPMD code of an example program.

Chapter 4

Experimental Evaluation

To evaluate the effectiveness of our integrated compiler and runtime techniques, we studied the performance of a set of HPF benchmark codes compiled with our SDSM-version of the dHPF compiler and executed on the enhanced TreadMarks SDSM system. We report results for four kernel benchmarks (Jacobi, RedBlackSOR, SOR and Tomcatv) and BT, an application benchmark from the NAS 2.0 Benchmark suite. For each of the benchmarks, we compare the performance of multiple configurations of compiler and runtime optimizations to ascertain the impact of these optimizations both collectively and individually.

Our experiments were performed on an IBM SP2. The SP2 is a distributed memory message-passing machine. Our experimental platform was populated with “high” processor nodes, each consisting of 4 PowerPC 604e processors with 1GB of main memory. Nodes are connected by a multi-layer scalable switching fabric. On each multiprocessor high node, we ran only one process to ensure that all messages were transported across the switch and that there was no contention for the network interface. Unless otherwise stated, all experiments were performed on 16 high nodes with one process active on each.

Our results are presented in similarly formatted tables, where the results of using different compiler and runtime configurations are shown. Table 4.2 describes information reported for each experiment and Table 4.1 shows our abbreviations for which optimizations were used in an experiment.

Opts	Meaning
–	barrier synchronization
P	point-to-point synchronization
E	eager update
R	restricted consistency
B	compiler-managed shared buffer

Table 4.1 : Key for row heading abbreviations in performance tables.

Columns	Meaning
Msg	number of messages delivered
UpMsg	number of eager-update messages sent
Comm	total amount of data communicated
Appl	amount of application data
Updt	amount of application data sent eagerly
Req	amount of application data sent in response to a request

Table 4.2 : Key for column heading abbreviations in performance tables.

4.1 Kernel Benchmarks

To evaluate the effectiveness of our strategy on loosely-coupled parallel computations, we applied it to two widely-used benchmark kernels: Jacobi and Red Black SOR (RBSOR). The Jacobi and RBSOR kernels are shown in Figures 4.1 and 4.2, respectively. In each phase, a set of array elements is updated from the values of a disjoint set of array elements. Since the values computed are disjoint from those used, there are no loop-carried data dependences. Thus, communication is necessary only at the end of each update phase. The results for the two benchmarks are qualitatively similar so we present only the Jacobi results in some detail.

To evaluate the our techniques for tightly-coupled computations, we studied the SOR kernel shown in Figure 4.3. In each phase, each array element is updated from its own neighbors. This computation has loop-carried data dependences. The dHPF compiler parallelizes this wavefront computation using coarse-grain pipelining which leads to tightly-coupled communication.

For each program we measured two problem sizes: 4Kx4K and 1Kx1K. For each program and problem size configuration, we measured the performance of a one-dimensional block computation partitioning in which each processor works on a block of columns known as a (*,BLOCK) partitioning,*. For the loosely-coupled benchmarks, we compare the 1D partitioning results with a 2D computation partitioning in which the computation is divided up so each processor works on one of 16 square tiles; this is known as a (BLOCK,BLOCK) partitioning.

4.1.1 Jacobi

Table 4.3 shows results for Jacobi with (*, BLOCK) 1D partitioning of a 1Kx1K problem on 16 processors. For this 1D partitioning, the combination of point-to-point synchronization, compiler-managed restricted consistency and eager update performed

*Since Fortran uses column major ordering for array elements, this means each processor works on a contiguous block of data.

```
do time = 1, maxtime
  do j = 2, m - 1
    do i = 2, n - 1
      a(i, j) = 0.25*(b(i-1,j)+b(i+1,j) + b(i,j-1)+b(i,j+1))
  do j = 2, m
    do i = 2, n
      b(i, j) = a(i,j)
```

Figure 4.1 : Jacobi computational kernel.

Opts	**Time (s)**	**Msg (K)**	**UpMsg (K)**	**Comm) (MB)**	**Appl) (MB)**	**Updt (MB)**	**Req (MB)**
-	12	91	0	190	184	0	184
P	12	136	0	198	184	0	184
PE	10	61	30	197	184	123	61
PR	9	106	0	134	123	0	123
PRE	7	31	30	134	123	123	0

Table 4.3 : Jacobi 16 processors, (*,BLOCK) partitioning, 1Kx1K elements. Sequential results 84s.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
-	188	12,470	0	12,820	12,710	0	12,710
P	336	12,850	0	12,860	12,720	0	12,720
PE	319	8,371	59	12,880	12,750	3,174	9,578
PR	150	5,771	0	6,455	6,393	0	6,393
PRE	169	2,912	59	6,472	6,402	3,174	3,228
PB	375	12,860	0	12,850	12,710	0	12,710
PEB	324	10,360	48	12,860	12,710	25	12,690
PRB	11	144	0	42	25	0	25
PREB	7	48	30	48	25	25	0

Table 4.4 : Jacobi 16 processors, (BLOCK,BLOCK) partitioning, 1Kx1K elements. Sequential results 84s.

40% faster than barrier synchronization. Restricted consistency reduced communication over a third, accounting for a 20% performance improvement. † Adding eager update accounted for the remaining improvement. Since a columnwise data partitioning does not lead to excessive fragmentation (the shared data is contiguous), the dHPF compiler does not transform the code to use compiler-managed communication buffers.

Next, we consider the (BLOCK,BLOCK) partitioning of the same 1Kx1K problem on 16 processors. Unlike the columnwise partitioning just described (in which there is little fragmentation), accessing a row of elements computed by a neighboring processor, which is necessary with a multi-dimensional partitioning, causes severe fragmentation. Unless both restricted consistency and a compiler-managed commu-

†For all 1024x1024 versions studied (barriers or point-to-point synchronization), the dHPF compiler automatically padded the arrays to 1025x1025 to reduce cache conflicts.

nication buffer are used to control the cost of this fragmentation, performance will be unacceptable.

Table 4.4 table shows a dramatic increase in time when moving from barrier synchronization (-) to point-to-point (P) synchronization without any other optimizations. The reason for this startling increase is that the asynchrony of point-to-point synchronization lets the processors computing corner tiles run ahead due to their lighter computation and communication loads. These processors keep interrupting their slower neighbors with many requests for fragmented pages, delaying their `signal` operations to other processors. These delays accumulate and spread causing a substantial overall slowdown. Adding restricted consistency (configuration PR) does nothing for the fragmentation for the `b` array, though it eliminates false sharing of the `a` array which cuts communication by 50%, cutting time in half from P alone. Adding just the buffer optimization to point-to-point synchronization (configuration PB) simply transforms fragmentation of the `b` array into false sharing, leaving communication volume virtually unchanged. However, applying restricted consistency and compiler-managed communication buffers together (configuration PRB) improves performance by nearly a factor of 15. Adding eager update to this mix (configuration PREB) reduces execution time by an additional 30%. Overall, the 2D partitioning with the PREB optimizations equaled the performance of the 1D partitioning showing that our optimizations can virtually eliminate any false sharing and fragmentation penalty of 2D distributions for loosely-synchronous computations.

Experiments with a 4Kx4K problem size yielded similar results, which are shown in Table 4.5 and Table 4.6 respectively. Moving from barrier synchronization to point-to-point synchronization with all optimizations with a 1D partitioning showed only a 6% improvement whereas for the 1Kx1K size the improvement was 40%. For this larger problem size, the higher computation/communication ratio diminishes the overall impact of any communication optimizations. The lower surface to volume ratio of the 2D partitioning began to pay off for the larger problem size, with execution

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
-	8.4	5.8	0	19	18	0	18
P	8.3	9.8	0	20	20	0	20
PE	7.9	2.6	2.4	20	20	20	0
PR	8.5	9.6	0	21	20	0	20
PRE	7.8	2.6	2.4	21	20	20	0

Table 4.5 : Jacobi: 16 processors, (*,BLOCK) partitioning, 4Kx4K elements. Sequential time is 115s.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
-	224	3,653	0	8,257	8,188	0	8,188
P	590	3,989	0	8,594	8,521	0	8,521
PE	450	1,974	19	9,419	9,345	4,035	5,310
PR	388	2,628	0	6,061	6,020	0	6,020
PRE	96	665	19	6,699	6,658	4,035	2,623
PB	846	2,631	0	5,444	5,382	0	5,382
PEB	710	2,624	3.8	5,467	5,381	7.9	5,347
PRB	8.5	12	0	9.3	7.9	0	7.9
PREB	7.6	3.9	3.8	9.8	7.9	7.9	0

Table 4.6 : Jacobi 16 processors, (BLOCK,BLOCK) partitioning, 4Kx4K elements. Sequential results 115s.

```
C     compute red points
           do j = 3, m - 1, 2
              do i = 3, n - 1, 2
                 a(i, j) = 0.25*(a(i-1,j)+a(i+1,j)+a(i,j-1)+a(i,j+1))
           do j = 2, m - 1, 2
              do i = 2, n - 1, 2
                 a(i, j) = 0.25*(a(i-1,j)+a(i+1,j)+a(i,j-1)+a(i,j+1))
C     compute black points
           do j = 3, m - 1, 2
              do i = 2, n - 1, 2
                 a(i, j) = 0.25*(a(i-1,j)+a(i+1,j)+a(i,j-1)+a(i,j+1))
           do j = 2, m - 1, 2
              do i = 3, n - 1, 2
                 a(i, j) = 0.25*(a(i-1,j)+a(i+1,j)+a(i,j-1)+a(i,j+1))
```

Figure 4.2 : Red black SOR computation loops.

time for the PREB configuration reducing execution time 8% over the fastest time for a 1D partitioning.

4.1.2 RBSOR

Our results for RBSOR is qualitatively similar to the results obtained for Jacobi. They are shown in Table 4.7, 4.8, 4.9, and 4.10. For instance, in Table 4.10, we present the results of running a 4Kx4K problem with a 4x4 (BLOCK, BLOCK) partitioning on 16-processors. It shows that the compiler controlled consistency cannot effectively combat fragmentation without the shared buffer optimization. Note that PRE performs significantly worse than PR though they are transferring almost the same amount of data. The reason is that system limits prevent one from pushing

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
P	9	146	0	212	194	0	194
PE	7	50	48	212	196	196	0

Table 4.7 : RBSOR 16 processors, (*,BLOCK) partitioning, 1Kx1K elements. Sequential result 54s.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
P	418	9,820	0	10,360	10,220	0	10,220
PE	413	8,378	77	10,950	10,220	590	9,630
PR	422	10,930	0	10,360	10,220	0	10,220
PRE	407	9,185	77	10,820	10,220	590	9,630
PB	413	10,360	0	10,250	10,090	0	10,080
PEB	366	8,311	77	10,250	10,080	39	10,040
PRB	12	232	0	68	39	0	39
PREB	7	79	77	77	39	39	0

Table 4.8 : RBSOR 16 processors, (BLOCK, BLOCK) partitioning, 1Kx1K elements. Sequential result 54s.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
P	8	19	0	41	39	0	39
PE	7	5	4.8	41	39	39.31MB	0

Table 4.9 : RBSOR 16 processors, (*,BLOCK) partitioning, 4Kx4K elements. Sequential result 94s.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm) (MB)	Appl) (MB)	Updt (MB)	Req (MB)
P	1520	3,438	0	10,770	10,690	0	10,690
PR	1206	3,952	0	10,740	10,660	0	10,660
PRE	1823	1,696	32	11,330	10,760	6,484	4,740
PB	529	2,515	0	5,187	5,123	0	5,123
PEB	545	2,626	7.6	5,473	5,383	16	5,367
PRB	8	23	0	19	16	0	16
PREB	7	7.9	7.6	20	16	16	0

Table 4.10 : RBSOR 16 processors, (BLOCK, BLOCK) partitioning, 4Kx4K elements. Sequential result 94s.

too much data at a time. If fragmentation is not optimized away with shared buffers and compiler controlled consistency, each processor ends up pushing 8MB of data to each of its row-wise neighbors in each iteration, thus saturating the message passing system and causing communication to be serialized. Thus, either the compiler or SDSM runtime library needs to throttle the amount of data that can be pushed at each synchronization point.

4.1.3 SOR

Like Jacobi, the SOR program shown in Figure 4.3 performs a stencil computation; however, the elements updated are in the same array as the elements being read. Since there are carried dependences on both the inner and outer loops, a multi-dimensional partitioning will provide less parallelism than simple 1D partitioning. Thus, we considered only a (*, BLOCK) column-wise partitioning.

Communication overhead increases as the granularity of the pipeline communication decreases and as the number of processors in the pipeline increases. As the

```
      do i = 2, n - 1
         do j = 2, m - 1
            a(i, j) = 0.25*(a(i-1,j)+a(i+1,j)+
     *                a(i,j-1)+a(i,j+1))
         enddo
      enddo
```

Figure 4.3 : SOR computation loop.

Opts	**Time (s)**	**Msg (K)**	**UpMsg (K)**	**Comm (MB)**	**Appl (MB)**	**Updt (MB)**	**Req (MB)**
P	29	903	0	118	49	0	49
PE	25	390	390	290	52	52	0

Table 4.11 : SOR 16 processors, (*,BLOCK) partitioning, 1Kx1K elements. Sequential time 56s.

number of processors increases, we need more pipeline stages to cover the startup and shutdown cost of the pipeline. Thus, for the 16 processor runs, the pipeline granularity (an option to the compiler) was chosen so each processor executes 64 pipeline stages, and for the 8 processor runs, the pipeline granularity was set for 32 pipeline stages.

Tables 4.11, 4.12, and 4.13 present 16-processor results for problem sizes of 1Kx1K, 4Kx4K and 8Kx8K. Speed-ups measured were 2 on 1Kx1K problem and 11 on 4Kx4K problem. Experiments with an 8Kx8K problem size achieved 13. Since fragmentation and false sharing are not an issue with this 1D partitioning, only eager update improves the performance. Configurations with other optimizations are not shown. The poor speedup on the 1Kx1K problem size illustrates the importance of a low communication to computation ratio which is obtained by using a sufficiently large

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
P	13	134	0	37	20	0	20
PE	9	39	39	42	20	20	0

Table 4.12 : SOR 16 processors, (*,BLOCK) partitioning, 4Kx4K elements. Sequential time 96s.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
P	19	64	0	18	11	0	11
PE	15	20	20	22	11	11	0

Table 4.13 : SOR 16 processors, (*,BLOCK) partitioning, 8Kx8K elements. Sequential time 187s.

pipeline granularity. It shows that small tightly-coupled problems are not appropriate for SDSM.

The SOR results are consistent with the observation that the parallel performance of a pipelined loop is determined by a number of factors. Above all, the computation to communication ratio must be large enough to dominate communication overheads. On the other hand, there must be enough pipeline stages to amortize startup and shutdown costs of the pipeline. Simultaneously satisfying both requirements requires large problem sizes. One difficulty is that even though there is little or no false sharing, the SDSM still attempts to maintain a globally consistent view of memory, so consistency meta-data is propagated across the entire length of the pipeline and back. As the number of processors and pipeline depth increases this becomes a significant cost. Eager update helps pipelining and mitigates cost of sending meta-data by avoiding round-trip communication, but this does not solve the problem.

Opts	Time	Msg	UpMsg	Comm	Appl	Updt	Req
P	32	284	0	45	23	0	23
PE	22	93	92	54	24	24	0

Table 4.14 : SOR 8 processors, (*,BLOCK) partitioning, 1Kx1K elements. Sequential time 56s.

Opts	Time	Msg	UpMsg	Comm	Appl	Updt	Req
P	20	33	0	12	9	0	9
PE	14	9	9	12	9	9	0

Table 4.15 : SOR 8 processors, (*,BLOCK) partitioning, 4Kx4K elements. Sequential time 96s.

Table 4.14, Table 4.15, and Table 4.16 present the results on 8 processors for the same problems as above. We obtained similar results on 8-processor runs, though the relative speedup figures are better because communication granularity is coarser on a smaller number of processors leading to higher efficiency.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
P	31	17	0	6	5	0	5
PE	27	5	5	6	5	5	0

Table 4.16 : SOR 8 processors, (*,BLOCK) partitioning, 8Kx8K elements. Sequential time 187s.

4.1.4 Tomcatv

This SPEC benchmark is primarily a loosely-coupled computation with reductions between phases. It uses a (BLOCK, *) partitioning by rows, so the performance using SDSM suffers due to severe fragmentation. However, with our optimizations, we achieved a speedup of 17 on a 16-processor run for a 1Kx1K problem instance. Our speedup of a 514x514 problem instance is 8, which more than double the speedup of 3.6 out of 16 reported by Han, Tseng, and Keleher [12]. Based on results reported by Anderson, Amarasinghe, and Lam [5], we expect that data reshaping could yield us significant further improvements by reducing TLB misses and cache misses. We have a prototype reorganizer, but it is not yet integrated with our SDSM compiler.

4.2 The NAS-BT Parallel Benchmark

NAS-BT is a large, "whole application" benchmark. Our parallel version was constructed by starting with the sequential version of the BT benchmark from the NAS suite [6], adding HPF directives, and interchanging a few loops to adjust the pipeline granularity exploited by dHPF. (These changes are described elsewhere [2].) Also, we inlined some function calls to improve data dependence analysis so the compiler could hoist synchronization out of inner loops. In different phases, NAS-BT exhibits the data reference patterns found in both Jacobi and SOR. For those phases in which computation sweeps along a partitioned dimension the compiler generates pipelined parallelism. By using point-to-point synchronization, this pipelining becomes a wavefront.

We ran the NAS-BT experiments on the class A problem size using (*, BLOCK) and (BLOCK, BLOCK) data distributions and on 4, 8, and 16 processors. The original sequential version runs in 3948 seconds‡.

‡Because of hardware differences, the results presented here are not directly comparable with the results presented in [2].

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
P	918	6,615	0	17,063	16,849	0	16,849
PE	599	617	441	17,629	17,335	16,785	550
PR	918	6,491	0	16,932	16,656	0	16,656
PRE	591	474	441	17,687	17,357	17,357	0.5

Table 4.17 : NAS-BT class A, 16 processors, (*, BLOCK) partitioning.

Table 4.17 summarizes 16-node runs column-wise 1D partitioning. There is not much false sharing, so the optimizations addressing false sharing give only modest improvements. The message aggregation and latency avoidance of the eager update mechanism does provide significant improvement. The 1D partitioning leads to longer pipelines (which increase serialization) which degrades efficiency as more processors are added.

Table 4.18 summarizes results for (BLOCK,BLOCK) 2D partitioning on 16 processors. False sharing and fragmentation are extensive. Compiler restricted-consistency by itself (PR) hides some of the false sharing, reducing the communication volume by a third, but it does not address communication latency or fragmentation. In contrast with the results for Jacobi and RBSOR, for NAS-BT compiler-managed buffers by themselves reduce the number and volume of messages. In the other two benchmarks there are accesses to the fragmented pages other than to the values transmitted in the buffers. Thus the pages remain falsely shared and incur consistency operations. In NAS-BT, this is a slightly less frequent occurrence. The added cost of buffer copying, however, means that execution time does not show a large decrease.

The combination of compiler-managed buffer and compiler restricted-consistency (PRB) works as intended to convert highly fragmented pages into falsely shared ones and to then hide the pages from the consistency mechanism. This eliminates two

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
P	926	6,320	0	14,904	14,702	0	14,702
PE	737	3,231	228	15,052	14,808	7,587	7,220
PR	916	4,674	0	11,143	10,941	0	10,941
PRE	595	754	232	11,334	11,029	9,879	1,213
PB	1066	6,425	0	10,689	10,458	0	10,458
PEB	852	3,421	221	10,795	10,461	4,891	5,569
PRB	585	3,252	0	5,075	4,931	0	4,931
PREB	405	247	220	5,151	4,931	4,891	40

Table 4.18 : NAS-BT class A, 4x4 processor array.

thirds of communication and reduces execution time by over 56%.

Used with any combination of the other mechanisms, eager update decreases communication cost through message aggregation and latency elimination. The best speedup, 10 out of 16, is achieved by applying all of the optimizations (PREB). Once the false sharing and fragmentation problems have been dealt with, the increased parallelism and smaller communication/computation ratio of the 2D partitioning contribute to better overall performance than is achieved using 1D partitioning as the results in Table 4.17 show.

We illustrate the timings for each of the computation phases of a 16 processor run of NAS-BT with 2D partitioning in Table 4.19. The row labeled *Seq* is the sequential execution times. Since we are doing 2D partitioning, both the *y_solve* and the *z_solve* use pipeline parallelism on one of the distributed dimensions, but the pipeline granularity is very fine-grained. The reason is that we have to keep a sufficient number of pipeline stages and there are not enough number of iterations in the pipelined loop to achieve good pipeline granularity. The best speedup for the

Opts	compute_rhs	x_solve	y_solve	z_solve	add
Seq	1212	735	893	988	117
P	279	62	234	260	7.5
PE	197	61	204	180	7.6
PR	263	57	251	286	7.4
PRE	161	57	141	176	7.4
PB	412	70	311	245	6.7
PEB	336	77	206	175	6.7
PRB	127	44	196	206	5.9
PREB	85	44	126	138	5.9

Table 4.19 : Timing of NAS-BT class A, 4x4 processor array.

pipelined loop such as in *y_solve* and *z_solve* is only about 7.2. However, we can see that applying all of the optimizations improved the performance of these pipelines by a factor of about 2 each. We achieved almost linear or super linear speed up for other parallel loops such as *compute_rhs*, *x_solve* and add. The computation pattern for these loops are very similar to Jacobi. Our success in achieving good speedup with NAS-BT shows that our integrated compiler-runtime SDSM is feasible in practice.

Table 4.20 and 4.21 are results from the 8-processor run and 4-processor run respectively. They both uses (BLOCK, BLOCK) partitioning. The results are qualitatively similar to the results shown in Table 4.18 for the 16-processor run. We achieve better speedup for smaller number of processors.

4.3 Overall Summary

After the evaluation of our experimental results, we summarize as follow.

The multi-dimensional data distribution and computation partitioning helps reduce the communication/computation ratio and increase the parallelism when com-

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
PB	1205	3,338	0	5,684	5,607	0	5,607
PEB	992	1,519	131	5,694	5,608	3,260	2,348
PRB	835	1,973	0	3,332	3,278	0	3,278
PREB	635	152	130	3,337	3,278	3,260	18

Table 4.20 : NAS-BT class A, 2x4 processor array.

Opts	Time (s)	Msg (K)	UpMsg (K)	Comm (MB)	Appl (MB)	Updt (MB)	Req (MB)
PB	2223	2,420	0	4,141	4,106	0	4,106
PEB	1935	1,475	63	4,145	4,108	1,630	2,478
PRB	1251	1,050	0	1,780	1,761	0	1,761
PREB	1053	106	63	1,781	1,761	1,630	131

Table 4.21 : NAS-BT class A, 2x2 processor array.

bined with techniques such as coarse grain pipelining. However, it usually can not achieve better performance on SDSM unless we use the compiler-restricted consistency and compiler-managed buffers optimization techniques to reduce the cost of false sharing and fragmentation across the distributed data dimensions other than the slowest varying one. The compiler-restricted consistency optimization also helps boost the performance even when only single-dimensional data distribution and computation partitioning are used.

The eager update of data at synchronization points can increase performance, when the amount of update data are controlled. The advantage of this techniques relies on transferring bulk data and avoiding the round trip communication. It helps utilize the available communication bandwidth and reduce the communication overhead of SDSM even though the total amount of communication data is unchanged or maybe increased slightly.

The point-to-point synchronization with compilation for coarse grain pipelining achieves impressive performance results for tightly-coupled applications on SDSM. However, good performance is contingent on a number of factors. These factors include a good computation/communication ratio to cover the SDSM runtime cost, sufficient pipeline depth to cover the starting and trailing cost of pipeline, and enough communication granularity to tolerate communication and SDSM runtime overhead.

Chapter 5

Related Work

The related work on improving SDSM performance falls into two categories: incorporating message-passing directly into SDSM, and reducing false-sharing/fragmentation in SDSM.

Incorporating message passing directly into SDSM systems is expected to be efficient and scalable, since there is no consistency data using this model for regular computations. However, to switch from regular computation phase to a more dynamic and runtime-dependent irregular computation phase, changes made by individual processors have to be merged and broadcast to other processors. Since there is no consistency meta-data, it is not easy to know which processor has the most recent values for shared pages that have been modified. There is a lack of research and study addressing this problem so as to provide a uniform message passing SDSM platform for applications that have both regular and irregular computation. One way to make this possible is to apply communicated data to overwrite both the virtual shared address and twins during the regular computation phases, which is not efficient. Redundant communication is also caused by the lack of consistency information. Processors should not write to the same piece of data; otherwise, consistency data has to be maintained to distinguish the order in which they were written. In this case, a compiler has to be exact about section analysis, since there is no diff creation to distinguish the actual data to be communicated at runtime. Furthermore, this approach does not provide a hybrid communication model as in our approach to adapt to the capacity of the communication system. Since all the data is going to be pushed at the synchronization point, it does not work well in the case where the communication is bursty and significant.

Two groups have studied incorporating message passing into SDSM directly. Mirchandaney, Hiranandani, and Sethi [25] proposed *DSM_Send* and *DSM_Recv* primitives for SDSM, which falls into the direct message passing category; however, their performance evaluation was incomplete which made it impossible to assess the merits of their approach. Dwarkadas *et al.* [9] proposed a compiler-directed *Fetch_Diffs* operation and a *Push* operation for SDSM. Their fetch primitive has each client request the data it needs at a synchronization sink using bulk data transfers. Unlike our eager update protocol, Fetch uses round trip communication to request data and can introduce extra wait time due to blocked signals. Their push operation is also a direct message passing incorporation approach. However, it is based on a global barrier synchronization, and only the synchronous version is supported. In addition, the regular sections used in their work are limited to a contiguous range of memory.

In comparison with direct message passing incorporation, our restricted consistency model maintains the meta data of the SDSM runtime at all times, but relies on compiler analysis to reduce the amount of runtime overhead and false sharing. It provides a smooth transition between regular part and irregular computations. As long as the communication granularity is sufficiently large to dwarf the cost of consistency data, this model can achieve very good performance.

There are many other approaches to reduce the effects of false sharing. The application programmer or compiler can add padding to the lower array dimensions to make the distributed array dimensions page aligned. However, data will be more scattered onto more pages resulting in more messages, even though the total amount of data transfered is reduced. The number of diff operations will also increase dramatically, which can drag down performance significantly. Cache conflict misses also increase because the array dimension size are made even and references within same loops are more likely mapped to the same cache lines. Furthermore, padding cannot fix the fragmentation problem caused by the non-contiguous array data references.

Reducing the sharing granularity by reducing the page size can reduce the amount of false sharing and fragmentation. However, the runtime overhead increases as well. Using a smaller page granularity also results in excessive misses and poor communication bandwidth. Zhou and Iftode etc. [31] have investigated using smaller coherence granularities of cache blocks for shared memory consistency, which uses a sequential consistency model and the Stache protocol [27] that is similar to directory based hardware implementations (e.g., [21]). However, it relies on the assistance of special expensive hardware assist.

Chapter 6

Conclusions

Our research study and experimental evaluation prove the thesis present in this document. That is with proper compiler support and sophisticated runtime implementation, an integrated compiler and runtime SDSM can achieve satisfactory performance for the parallelization of a range of applications.

Our experiments demonstrate that our integrated compiler and runtime support for SDSM is very effective at reducing communication and runtime overhead associated with consistency maintenance and page fault handling. We have shown that the combination of compiler-controlled restricted consistency in conjunction with compiler-managed communication buffers are very effective at reducing the amount of false sharing and fragmentation in SDSM. Compiler-restricted consistency and compiler-managed communication buffers avoid maintaining full consistency for a large number of pages when computation is partitioned along a data dimension of a multi-dimensional array other than the slowest varying one. Bulk-data transfer associated with eager update usually boosts performance, as long as an excessively large quantity of data is not pushed at one time. The amount of data pushed at a signal operation can be capped so that any data above a selected threshold will instead be lazily requested by the consumer.

For applications with wavefront parallelism, our coordinated compiler and runtime support for point-to-point synchronization is very effective at exploiting wavefront parallelism with coarse-grain pipelining. By reducing false sharing and fragmentation through our compiler-directed point-to-point synchronization, scalable multi-dimensional computation partitioning becomes feasible with SDSM. Applications

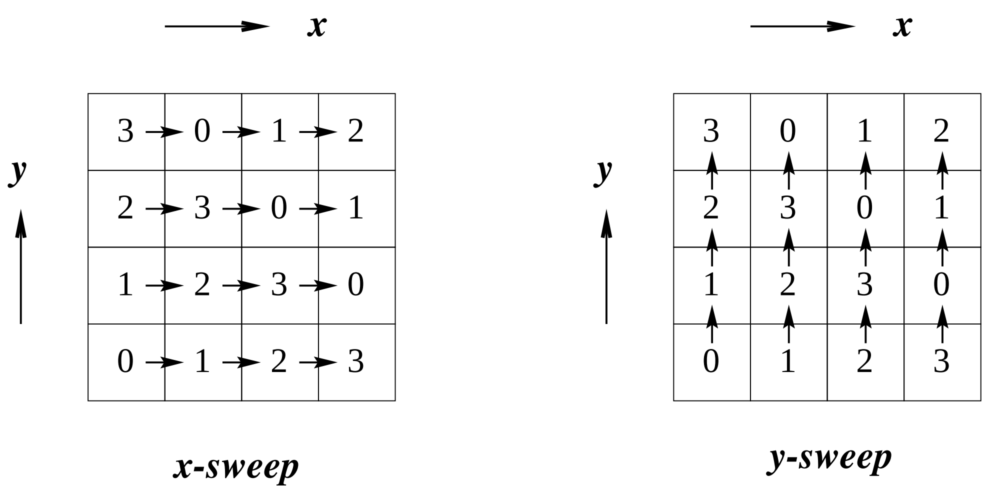

Figure 6.1 : Multi-partitioning

such as the NAS BT application benchmark can benefit substantially from multi-dimensional partitioning which increase the efficiency of wavefront parallelism.

Our experience with large benchmarks shows that applying compiler knowledge to optimize SDSM is a practical thing to do and can help achieve good performance for regular applications, even those with carried data dependences that force tightly-coupled data sharing. We believe that our experience shows that SDSM can be used effectively for achieving modest scale parallelism.

A significant result of our work is that we successfully parallelized the NAS-BT benchmark with SDSM and boosted the speedup on a 16-processor run from 4 to 10, when applying the compiler/runtime optimization techniques described in this document. The reason why we have not achieved even better performance is that the communication granularity in our coarse-grain pipelining is too small to compensate the overheads associated with maintaining and communicating consistency meta-data based on pages, such as diffing. We expect to improve speedup dramatically by using multi-partitioning [26], a skewed cyclic distribution that assigns each processor a set of tiles along a diagonal of a partitioned array so that a processor owns a piece of every row and column. With this partitioning, a processor has a block of data on which to compute during each computation phase of a directional sweep. Figure 6.1 illustrates the computation partitioning when using a multi-partitioning distribution. Such a distribution is very useful for orthogonal computation sweeps such as those found in NAS-BT. Multi-partitioning uses only a coarse-grain communication. In each phase, each processor computes results for a tile, then all processors perform a communication step. By using multi-partitioning with our existing optimization techniques, we expect to avoid the overheads caused by fine-grain communication that can arise with pipelined computations.

In closing, however, we note that for SDSM to be an effective platform for large-scale parallel computing, limitations in SDSM scalability will require further work. Our experiences with very large data benchmarks exposed a number of SDSM implementation shortcomings, such as requiring all consistency information at a barrier to be transmitted in a single message, that will need to be addressed.

Bibliography

[1] S.V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proceedings of the Second High Performance Computer Architecture Symposium*, pages 26–37, February 1996.

[2] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[3] Vikram Adve and John Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[4] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[6] D. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-

020, NASA Ames Research Center, December 1995.

[7] R. M. Beam and R. F. Warming. An implicit factored scheme for the compressible navier stokes equations. 16(4):393–402, 1978.

[8] S. Chandra and J. Larus. Optimizing communication in HPF programs on fine-grain distributed shared memory. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 100–111, June 1997.

[9] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, October 1996.

[10] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.

[11] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[12] Hwansoo Han, Chau-Wen Tseng, and Pete Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International Journal of Parallel Programming*, 26(5):591–612, October 1998. Invited paper from LCPC'97.

[13] Claes Johnson. *Numerical solutions of partial differential equations by the Finite Element Method*. Cambridge University Press, 1987.

[14] Claes Johnson and Anders Szepessy. On the convergence of a finite element method for a nonlinear hyperbolic conservation law. *Mathematics of Computation*, 49(180):427–444, 1987.

[15] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[16] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 91–98, May 1996.

[17] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, October 1995.

[18] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[19] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[20] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, April 1996.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[22] K. Li. Ivy: A shared virtual memory system for parallel computing. *Proceedings of the 1988 International Conference on Parallel Processing*, 2:94–101, August 1988.

[23] Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, March 1998.

[24] H. Lu, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Software distributed shared memory support for irregular applications. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 48–56, June 1996.

[25] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of software DSM systems via compiler involvement. In *Proceedings of Supercomputing '94*, 1994.

[26] V. Naik. Scalability issues for a class of cfd applications. In *Proceedings of Scalable High Performance Computing Conference*, pages 268–275, 1992.

[27] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[28] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[29] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[30] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, nov 1996.

[31] Y. Zhou, L. Iftode, K. Li, J.P. Singh, B.R. Toonen, I. Schoinas, M.D. Hill, and D.A. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, jun 1997. To appear.