Appears in the *Ninth Workshop on Languages and Compilers for Parallel Computing*, August 1996.

HPF on Fine-Grain Distributed Shared Memory: Early Experience

Satish Chandra and James R. Larus

Computer Sciences Department
University of Wisconsin—Madison
1210 W. Dayton Street
Madison, WI 53706 USA
{chandra,larus}@cs.wisc.edu
July 29, 1996

Abstract. This paper examines the performance of a suite of HPF applications on a network of workstations using two different compilation approaches: generating explicit message-passing code, and generating code for a shared address space provided by a fine-grain distributed shared memory system (DSM). Preliminary experiments indicate that the DSM approach performs with usually a small slowdown compared to the message passing approach on regular programs, yet enables efficient execution of non-regular programs.

1 Introduction

High Performance Fortran (HPF) [20] is the product of many years of collective experience with compiling for distributed memory machines. Researchers and companies have built compilers that compile HPF-like languages to efficient message-passing code [7, 13, 15, 19, 29, 34]. Yet, the domain of programs for which such compilers generate efficient code is very limited: good results have been demonstrated only on *regular* programs. Programs that use complicated array subscripts, such as those in the Perfect Club benchmark suite [11], have not been successfully compiled for message-passing machines. Consequently, despite HPF's allure, compilers limit the parallel applications that can benefit from HPF.

A compiler targeting a message-passing machine converts parallel loops that manipulate data in a global address space (such as those that can be written in HPF or similar languages [19,34]) into SPMD code that, in essence, synthesizes a global name space using explicit messages [22]. Unfortunately, the compiler depends on complete and accurate program analysis [34] to generate good message-passing code. Programs that cannot be completely analyzed show poor performance [29].

An alternative approach leaves the onerous task of implementing a program's shared address space to an underlying system. With an underlying coherent shared address space, compilers can greatly expand the range of programs that they can compile efficiently, as demonstrated by the Illinois Polaris [26] and the Stanford SUIF [36] compilers, which perform reasonably well for many non-regular programs. Shared address space at the system level can also aid HPF programmers in another significant way, as many large programs could occasionally go into an explicit task parallel mode, while accessing the same data set (HPF's EXTRINSIC). For example, an FFT algorithm, often a component in larger problems, can be written much more efficiently using task parallelism [12]. Most programmers, when writing such extrinsic routines for HPF code, would prefer to find their arrays in shared memory, than broken up and renamed by the compiler in complicated ways.

The requirement for an underlying shared address space, moreover, need not limit our choice of platforms to hardware implemented shared memory, such as the Stanford DASH [23] multiprocessor. An attractive alter-

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, NSF Grant MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation, Sun Microsystems, and The Portland Group. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

native is to use a distributed shared memory (DSM) system, which implements a shared address space on top of a message-passing substrate using software, or a combination of hardware and software. Recently, researchers have demonstrated efficient implementations of fine-grain (i.e., coherence at 32-128 byte granularity) shared memory for message-passing machines. For example, the Blizzard system at Wisconsin implements, in software, coherent shared memory on a CM-5 [32], and on a cluster of workstations [31].

A key question is whether a compiler is justified in incurring the overheads of a DSM-based shared address space, in preference to low-level message passing. This paper reports an experimental study that explores this question. In this study, our platform is a cluster of SPARCstation 20 workstations connected by a Myricom Myrinet [5] network. A runtime interface, called Tempest [17], provides the primitives for communication, as well as support for shared memory implemented almost entirely in software. We modified a commercial HPF compiler—the Portland Group, Inc. (PGI) *pghpf*—to generate code for Tempest's message passing and for shared memory implemented with Tempest mechanisms. The result from our (admittedly limited) experiments is that fine-grain shared memory runs with usually a small performance degradation (0%-55% increase in execution time) compared to native message passing on regular programs, yet enables efficient execution of non-regular programs. It is important to bear in mind that our compiler and run-time system do not yet attempt DSM-specific optimizations, so the balance is likely to shift further in favor of fine-grain shared memory.

The rest of the paper is organized as follows. Section 2 discusses related work in compiling for message passing and shared memory, and briefly explains the limitations of both approaches. Section 3 describes the compiler infrastructure that we used in this study, and Section 4 describes the platform that we used to run our programs. Section 5 presents experimental results on seven benchmarks. Section 6 presents some discussion and Section 7 concludes the paper.

2 Compiling for Message Passing and Shared Memory

We first consider the problems that plague compilers for message-passing machines. In absence of a shared address space, such compilers must partition arrays into local chunks that reside in per-node memories and modify the computation to accommodate the segmented address space. Computation is usually divided using the owner-computes rule [29]. Compilers introduce explicit messages at non-local references. Since communication in this paradigm is sender-initiated, the owner of non-local data sends it to processor(s) that need it. A naive, but general technique is run-time resolution [29]. However, this scheme may cause a slow down of a factor of several hundred over uniprocessor runs. Most compilers, therefore, exploit the observation that loops with no loop-carried dependence can obtain all non-local values before executing a loop. This optimization, called message vectorization, is critical for good performance, because it replaces small, frequent messages by large, infrequent ones, permits partitioning of the loop bounds [34], and permits overlap of computation and communication. In the best case, a compiler can statically determine the non-local data requirements for each processor and place all communication outside loops. This static analysis, however, is difficult for codes that use complex array subscripts or complicated control flow. Blume and Eigenmann [4] found that many dense matrix codes in Perfect Benchmarks [11] involve such programming constructs. Even if a programmer guarantees absence of loop-carried dependence by an INDEPENDENT directive, compilers cannot always vectorize the communication (although in some cases, inspector-executor technique [19, 30] has been found useful).

Not surprisingly, the efficacy of message-passing compilation has been demonstrated only for programs that consist wholly of simple loops and simple array subscripts. Our base compiler, *pghpf*, tries to classify the communication in a parallel construct (based on array subscripts) as one of overlap-shift, section-copy, or scatter-gather, failing which, it generates scalar communication inside loops [6].

In cache-coherent shared memory, the underlying system takes responsibility of fetching the latest value of a reference, irrespective of whether it is local or remote. This simplifies the task of a compiler to spreading parallel loops among processors and inserting synchronization. However, experimental studies have shown that

to obtain good performance, compilers for shared-memory machines have to be aware of the features of the underlying memory system, such as finite cache size and false sharing. Several studies have proposed and implemented data and loop transformations to increase locality of reference [2, 3, 10].

Several techniques can be used to reduce the data access costs in shared memory systems. Write-misses can be made less costly by buffering them until a sychronization point. Weaker memory consistency models [1, 14], which suffice for compiler parallelized codes, allow this optimization. Read latencies can be alleviated by judicious use of prefetching, although compiler algorithms for prefetching [25] have been few. Mirchandaney [24] and Koufaty [21] suggest augmenting their shared memory systems with a send primitive, which could reduce coherence overheads in some cases. Finally, Tseng [35] has presented a compiler algorithm to eliminate redundant barriers in compiler parallelized shared memory programs, and to replace barriers by pairwise synchronization where applicable.

In this study, we performed none of these compiler-directed optimizations. Most of these optimizations require analyses similar to those needed for message passing, and while such analyses will certainly help shared memory performance, the goal of our current experiment is to compile without deep program analyses. However, we have used an implementation of the weaker consistency model for our shared memory experiments.

3 Compiler Infrastructure

We used PGI's *pghpf* compiler (version 2.0). *pghpf* is a nearly complete implementation of HPF (it implements more that the Subset HPF). The compiler translates the input HPF to a node Fortran program containing calls to a runtime library for message passing. The node program is then compiled with PGI's *pgftn* compiler. *pghpf* performs a number of standard optimizations, most importantly message vectorization. Good performance has been reported on regular applications [6].

Our message-passing version is a straight-forward modification of *pghpf*'s runtime system to use Tempest messages. We use asynchronous transfer (with receiver side buffering) for small messages (up to 4k bytes), and synchronous unbuffered transfer for larger messages. The shared memory version involved several changes to the compiler. All distributed arrays are allocated in shared memory; replicated arrays are allocated in per-processor private memory. The compiler generates accesses to distributed arrays by their global names rather than the local equivalents. Parallel loops (arising from array statements, FORALL statements, and the INDEPENDENT directive) are separated by barriers. All statements and control structure outside parallel loops execute on all processors, except assignments to shared data, which are protected by a guard. The division of computation for parallel loops is still owner computes, in that the data distribution directives are followed to assign work to processors. Parallel loops marked INDEPENDENT are currently distributed block wise by loop index. Unlike message passing, there is no need for explicit communication.

We also modified the *pghpf* runtime code for performing reductions from a binary tree scheme into a flat reduction scheme where one processor gathers the operands from all other processors, performs the reduction, and broadcasts the result. On our eight node experiments, we found that the flat scheme significantly outperforms the binary tree on all reduction-intensive benchmarks (e.g. *gravity*, Section 5.3).

4 Experimental Platform

Our message-passing and shared-memory platforms are built on the Tempest [17] system. Tempest is an interface that provides the mechanisms needed to implement fine-grain coherent shared memory. These mechanisms include: (1) active message style message passing, (2) fine-grain access control, (3) bulk data transfer for sending large messages (4) virtual memory mechanism to map pages from the shared data segment locally. The most remarkable feature of Tempest is the fine-grain access control, which allows coherence to be maintained at the level of small blocks of memory (e.g. 128 bytes is the block size used in our

experiments) in contrast to coherence at page granularity in DSMs such as Treadmarks [18]. Using these mechanisms, a coherence protocol can be written entirely in user-level software (as a library) and linked with an application; details on the implementation of a coherent protocol using Tempest mechanisms can be found elsewhere [27]. For the current set of experiments, our protocol implements a version of the weakly consistent memory model—it attempts to reduce write latency by not waiting for the write ownership grant from the home node. At synchronization points, the node waits for all pending transactions to complete. One noteworthy feature of our coherence protocol is that it uses a portion of a node's main-memory as a large level-three cache [27], for holding remote data (like COMA [33]); this alleviates the problem of throwing away expensively fetched remote data due to finite size of the level-two cache.

These experiments ran on a cluster of dual-processor SPARCstation 20 workstations running Solaris 2.4 connected by a Myrinet network (all commodity parts). This implementation (see [31] for details) uses a small custom hardware device [28] that sits on the memory bus of each workstation and accelerates access control functions. Note that the coherence protocol itself is written in unprivileged software. Purely software implementations of fine-grain access control also exist, but they generally perform slightly slower. We perform computation on only one processor of a workstation node, leaving the other for coherence protocol related tasks. Although one could use both the processors for computation, we believe that future workstations will routinely have 4 or more processors, and one of them could be spared for protocol processing without noticeable loss of compute power. Some details on various components of the system are summarized in Table 1.

Processor	66 MHz HyperSPARC (2)
Network Interface	Myricom's Myrinet
Minimum roundtrip latency for short (4 bytes) message	40 µs
Network bandwidth	20 MB/s
Read miss processing time for 128 byte block	93 µs

Table 1: Some details of the cluster configuration used.

5 Results

We present results on seven HPF applications, listed in Table 2 with their problem sizes and memory usage.

Application	Source of HPF version	Problem Size	Memory(Mb)
pde	Genesis. HPF by PGI	grid size 128, 40 iters	56
shallow	NCAR. HPF by PGI	grid size 513, 100 iters	14
gravity	HPF by Syracuse	grid size 128, 5 iters	17
lu	Stanford. HPF by authors	1024x1024 matrix	4
tomcatv	SPEC. HPF by PGI	grid size 257, 100 iters	4
trfd	Perfect. HPF by authors	n=50 (1275x1275)	51
lcp	HPF by authors	8k rows, 0.5%sparsity	4.5

Table 2: Application Suite

For each application, we describe the structure of the application and its communication pattern. We then report the execution times on shared memory and message passing for an eight node cluster, as well as the uniprocessor times. The uniprocessor versions of programs were single processor Fortran codes obtained from the HPF program—they contain *no runtime parallelism overhead*. The uniprocessor times come from a similar workstation node containing more (96M) physical memory so none of the applications page. The times for the message passing versions are decomposed into time spent in computation and time spent in the communication libraries. Likewise, times for the shared memory versions are decomposed into time spent in computation, and time spent handling remote misses and waiting at barriers. Time spent in any reductions is counted as communication time in both versions. The speedups curves are obtained by dividing the uniprocessor execution time by the execution times for 2, 4 and 8 nodes.

5.1 PDE

PDE performs red-black successive over-relaxation on a 3 dimensional grid. Accordingly, the main data structures in this program are three 3-dimensional arrays of double precision numbers. These arrays are distributed blockwise in their third dimension, on a linear arrangement of processors. The primary source of communication in this program is the shift operation in each dimension, which causes near-neighbor communication in the third dimension (as it is distributed blockwise). The program is very communication intensive, as in each iteration, $O(n^2)$ values are communicated for $O(n^3)$ computation. With a small data set and a relatively slow communication substrate, *PDE* shows moderate speedups for both message passing and shared memory. Figure 1 summarizes the results.

Shared memory is takes about 36% more time than message passing in this case. The poor performance can be attributed to our invalidation-based coherence protocol. Previous studies [9] have shown that producer-consumer sharing behavior performs poorly with such a coherence protocol. Briefly, the standard invalidation-based protocol requires four messages to transmit a single cache line from a producer to a consumer: (1) read request from consumer, (2) reply from the producer, (3) invalidate request from the producer, and (4) acknowledgment from the consumer. Our weak consistency protocol helps overlap the delay only in steps 3 and 4. Not surprisingly, given a program dominated by producer-consumer data transfer, transparent shared memory exacts a cost in performance.

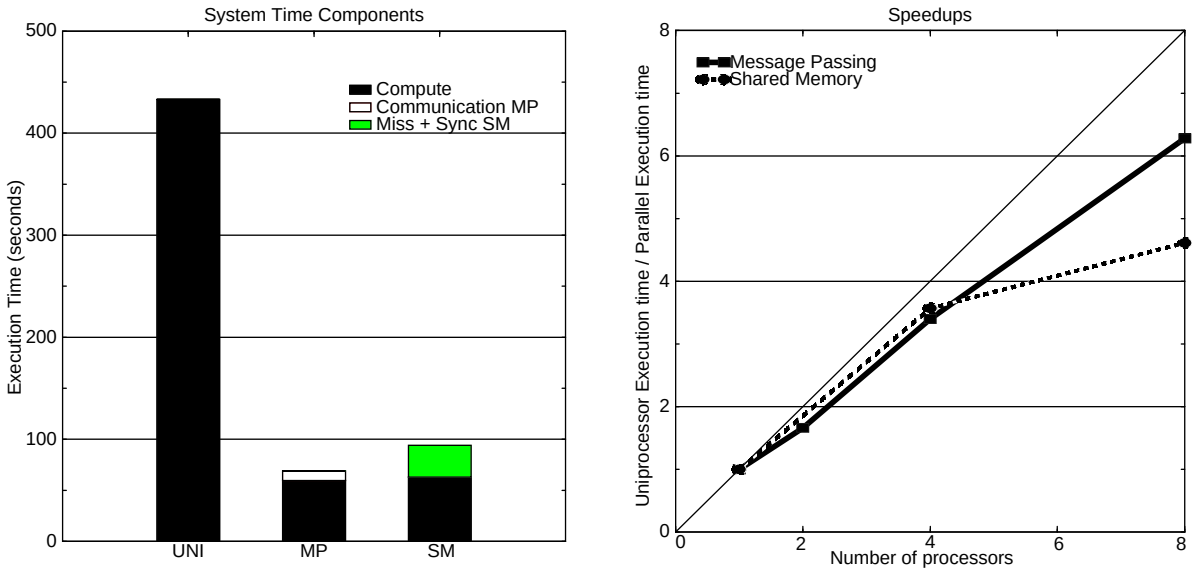

Figure 1: Performance data for *PDE*.

5.2 Shallow

Shallow has 14 2-dimensional arrays of single precision floating point numbers. All arrays are distributed by blocks of columns on a linear arrangement of processors. This program also performs shift communication in each dimension, causing near-neighbor communication in the second (distributed) dimension. Figure 2 presents the results.

As with *PDE* (Section 5.1), *Shallow* exhibits moderate speedups on both message-passing and shared-memory for this data-set size. A noteworthy characteristic of this application is that it exhibits message redundancy [16] across loop nests: values communicated for an earlier computation, in some cases, are still *available* (in the terminology of [16]) and need not be resent. The current version of PGI compiler does not perform message redundancy elimination of this kind. Shared memory, however, benefits from automatic caching as remote values are always available to consumers until new values are produced. In addition, *Shallow* can also benefit from message combining [8], an optimization applicable to both message passing and shared memory. Again, we have not explored this optimization yet. These observations apart, shared memory performs within 20% of message passing, even though all communication is producer-consumer.

5.3 Gravity

Gravity has two 3-d arrays and several 2-d arrays that are aligned with the last two dimensions of the 3-d arrays. In each outer time-step loop, the program iterates over the first dimension of the 3-d arrays, and performs computations on the 2-d plane formed by the second and third dimensions. These computations cause near-neighbor shift communication, and several SUM reductions in the same plane. We slightly modified the code in the mkl2 routine to work around a problem with the PGI compiler. Also, deviating from the original (*, BLOCK, BLOCK) distribution of the 3-d arrays, we used a (*, *, BLOCK) distribution; this was done to accommodate a problem with shared memory allocation in our current shared-memory compiler. Figure 3 presents the performance results.

Since this program executes reductions very frequently, both versions require an efficient mechanism. As noted earlier (Section 3), we used a flat reduction rather than a tree based reduction. In comparison with the tree reductions, the flat reductions took 44% less time for message passing and 35% less time for shared memory. Note that the shared memory reduction is not implemented solely using shared memory mecha-

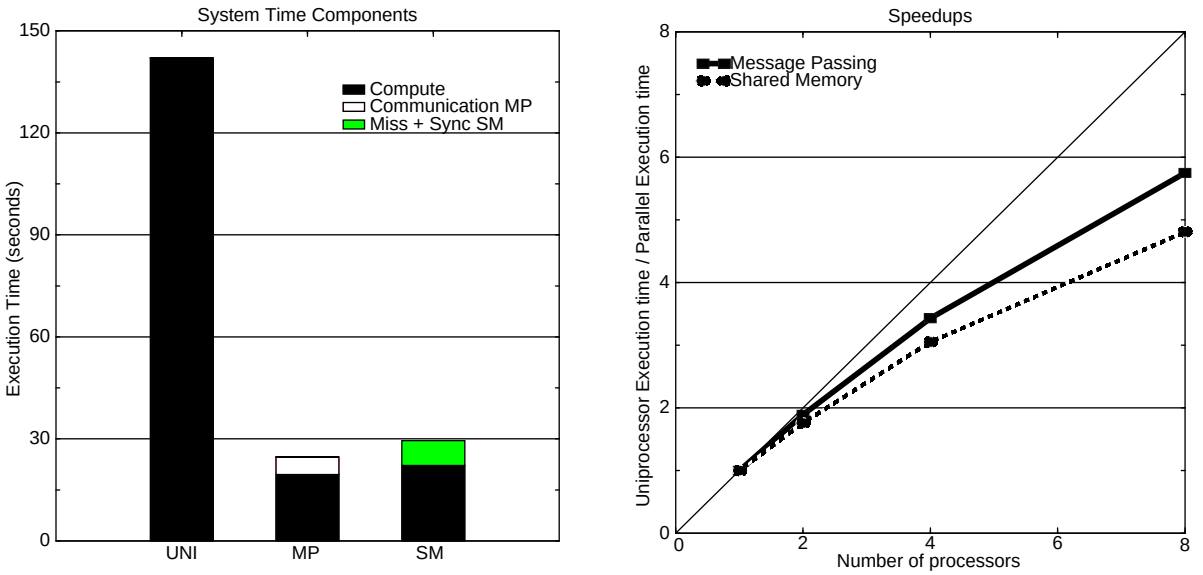

Figure 2: Performance data for *Shallow.*

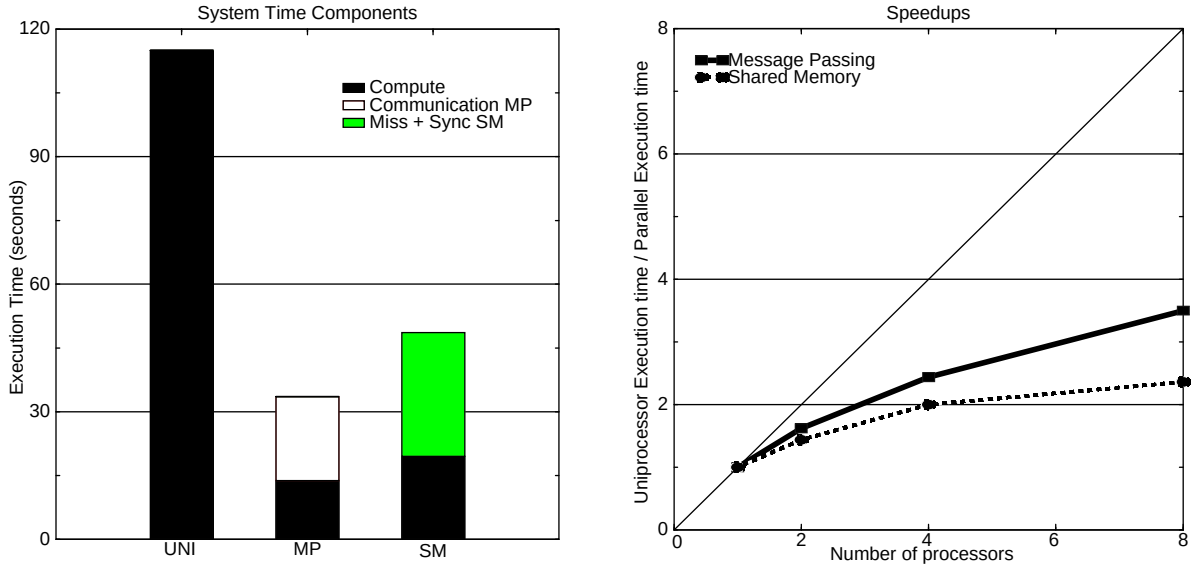

Figure 3: Performance data for *Gravity.*

nisms. With Tempest mechanisms, this optimization is straightforward, but we expect most DSM systems to provide lower level hooks for specialized tasks such as broadcasts and reductions. The remaining communication in the program is mostly near-neighbor. As in *Shallow* (Section 5.2), both shared memory and message passing versions can benefit from message combining [8]. However, for the current data set size, both message passing and shared memory versions spend a significant amount of time in communication, and do not show good speedups. Larger data set sizes (albeit with enormous memory requirements) are likely to increase the computation to communication ratio and achieve better speedups.

5.4 LU

LU performs LU decomposition on a dense matrix. The only data structure in the program is a two dimensional matrix that is distributed cyclically by columns, in order to maintain load-balance. The computation in *LU* is distinct from the previously presented grid-based programs. Each outer iteration of the program performs (sequentially) some computation on a pivotal column, and then subtracts a multiple of the pivotal columns from the remaining unprocessed column. The communication pattern, therefore, is a broadcast of a column vector from one node to all other nodes in each outer iteration. Figure 4 presents the performance results.

LU achieves a speedup of 6 on 8 nodes for the message passing version, which is good in view of the fact that the computation on the pivotal column is a sequential bottleneck in each iteration. The shared memory version does not perform as well. There are two reasons for this behavior. Since the memory allocation is column major, and the pages are distributed by blocks, each processor touches the whole virtual address space occupied by the 2-d array. As noted in [2], this exacts a cost in memory system performance, although, in contrast with [2], we do not suffer from replacement-to-home misses.

5.5 TOMCATV

Tomcatv consists of regular stencil operations on a number of 2-dimensional arrays. The distinguishing feature of *tomcatv* is that the parallelism is best exploited if the 2-dimensional arrays are distributed blockwise by rows rather than by columns. In contrast with a column-blocked distribution, the row-blocked distribution

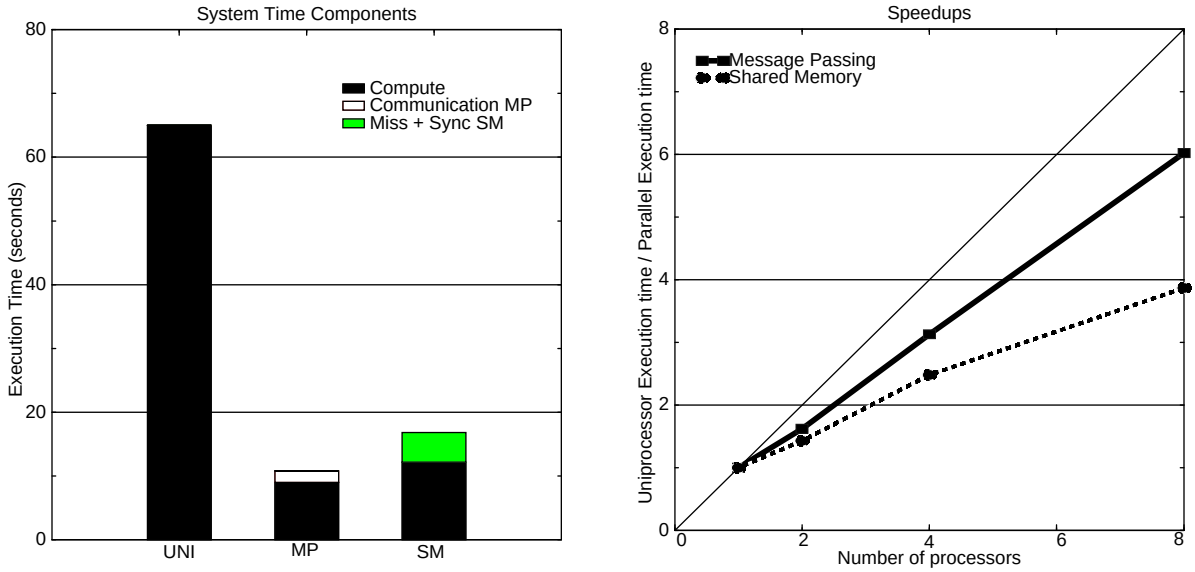

Figure 4: Performance data for *LU*

has a significant bearing on the shared memory performance. The communication in this program is primarily shift communication across the rows. Figure 5 shows the performance results.

Tomcatv does not perform well on either message passing or shared memory. The problem with this program arises from *pghpf*'s parallelization strategy. A key compute-intensive loop-nest in this code has several scalar variables that store the values of array expressions to be used multiple times, but within the same iteration of the inner loop. *pghpf* attempts to express all parallelism in a Fortran loop nest in terms of equivalent FORALL statements (in a later phase, it fuses loops with identical iteration distributions). Hence, it promotes all these scalar variables to arrays that match the main data arrays in shape and size. This results in increased local data access costs for message passing, as evidenced by poor speedup even in the compute cost (only 4.7 times on 8 processors). For shared memory, this increase in effective data set size is more taxing. Our data layout in shared memory is done by blockwise distributing the pages involved in an array's virtual address range. Since

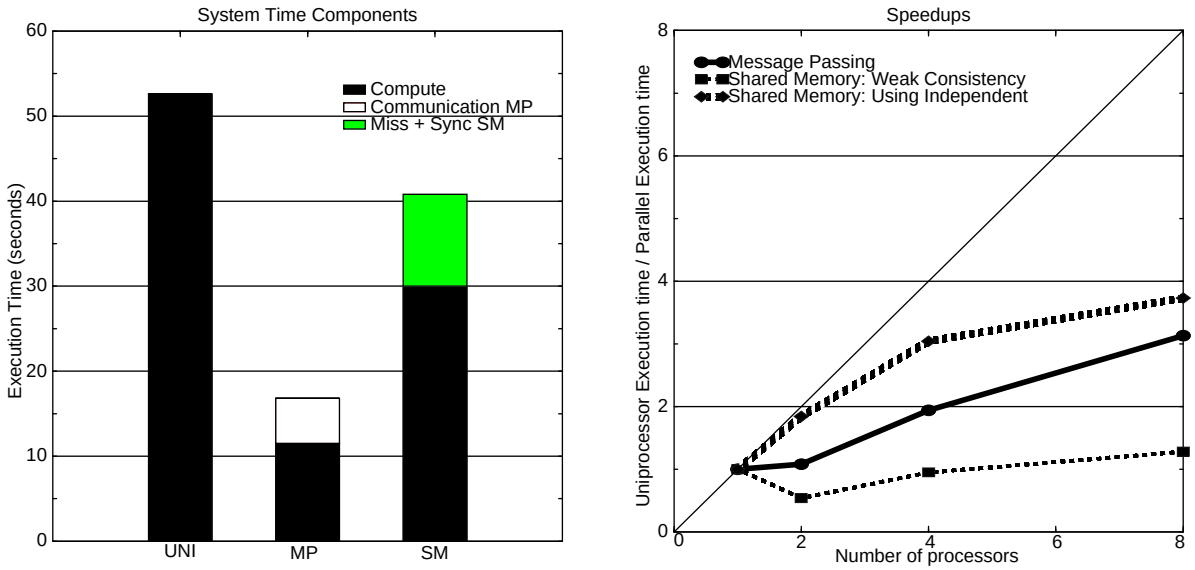

Figure 5: Tomcatv. Different versions of HPF source code are used for MP and SM (see text).

the computation decomposition touches blocks of rows instead of blocks of columns, each processor ends up bringing in entire data set locally. The arrays created by the compiler are aligned and distributed identically to the main data arrays, and exacerbate this problem. The same HPF code runs quite poorly on shared memory, taking about 930 seconds (an 18 times slowdown).

Following the observation in [2], we manually rewrote *tomcatv* for shared memory so it operates in a transpose fashion. Thus, each reference a(i,j) was converted to a(j,i), making all necessary changes in the program so it computes the same result. This is the shared memory version we used for the performance data reported in Figure 5. Shared memory performs much better with this version, yet, it fails to give any speedup: the slowdown for two processors is an indication of the high data access costs caused by temporary arrays. We were, however, able to create a shared memory version which did not require this temporary storage, by using the INDEPENDENT directive. For this version, the speedup obtained by shared memory matches, and even exceeds, that of message passing. On eight nodes, the compute portion of this version runs 5.8 times faster than the uniprocessor code. Unfortunately, we were unable to convince the compiler to produce a similar message passing version for a fair comparison.

5.6 TRFD

TRFD is a quantum mechanics kernel from the Perfect Club benchmark suite. The interesting feature of this program is that although it has considerable amount of loop level parallelism, the array subscripts are not affine functions of loop indices. Hence a compiler generating message-passing code has to resort to run-time resolution, thereby rendering parallel execution meaningless.

We produced an HPF version of *TRFD* by modifying the original code from Illinois in several significant ways. The original code has been written with assumptions of linear memory, in which several 2-dimensional arrays are carved out of a single large linear array. Linear memory is largely incompatible with HPF-style distributed arrays. Hence we declared the required distributed arrays explicitly (along with DISTRIBUTE directives), and modified the program to access those instead. Furthermore, actual subroutine arguments in HPF have to conform with their formal parameters in several ways (see [20], Chapter 5). We made the appropriate changes. Finally, in the original code, two of the loops that can be executed in parallel make a procedure call (TRANSF) in their loop body. Since we wanted to use the INDEPENDENT directive for these loops, we had to inline the procedure to form the loop body. All these modifications were necessary to obtain a legal and

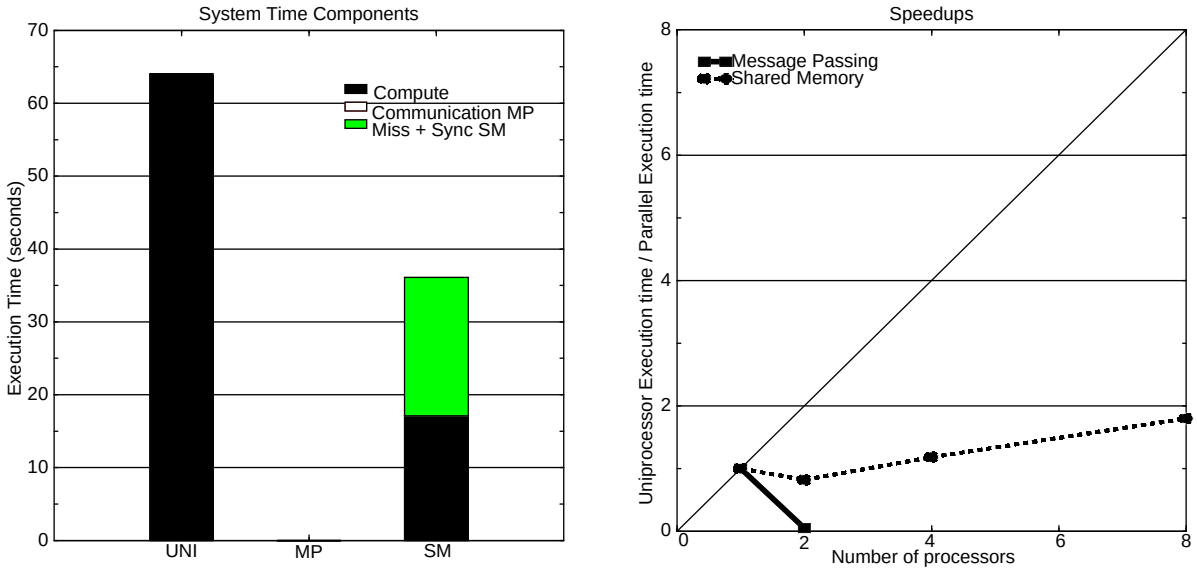

Figure 6: Performance data for *TRFD*. We could not run *TRFD* for more than 2 nodes.

efficient HPF program. The resulting HPF code manipulates a 2-dimensional array that is distributed by blocks of columns. The first loop, in subroutine INTGRL initializes the array using indirect accesses. Application-specific knowledge, or complicated dependence analysis [4] can reveal that the loop can indeed be executed in parallel. The other two loops, both of which have the TRANSF subroutine as their loop bodies, perform indirect operations only in the row dimension of the array, so all columns can be processed in parallel. Our shared memory compiler simply distributes these loops across processors (note that we use the INDEPENDENT directive). Figure 6 shows the performance numbers.

Although the program has sufficient high level parallelism, we were unable to cast the complete HPF code in terms of FORALL statements that would be accepted by the *pghpf* for efficient message passing. As a result, message passing version essentially performs the first loop sequentially, and generates calls to scalar communication in the other loops. Not surprisingly, the performance of the resulting message passing code is unacceptable: it shows a factor of 20 slowdown on a 2-node execution; we were unable to complete runs for higher number of nodes in any reasonable time. The speedup of 1.8 for shared memory, while not satisfactory, is a step towards efficiently executing non-regular programs written in HPF. This program demonstrates a case in which shared memory layer performs far better than direct message-passing code.

5.7 LCP

LCP solves the linear complementarity problem on a sparse system. The main obstacle that we encountered in writing the HPF code was the initialization. Since the input generation in done off-line, a large (4.4 Mb of binary representation) file had to be read in order to initialize all the arrays. Direct input in HPF, using an ascii version of the file, proved too slow. Instead, we declared a *shadow file* on each node, that was initialized with the contents of the input file by calling an external C function. The HPF code, then, simply reads off the values from the shadow files into distributed arrays. The data structures in the program consist of a sparse matrix represented by 3 arrays: an array containing the non-zero values *a*, an array containing the column index *ia* of each non zero element, and finally, and array marking the start of each new row *ja*. In addition, there is a global solution vector *xsol* that is updated once every few iterations until convergence is reached, and a local solution vector *xbar* used for intermediate values. In our implementation, we to distributed *a*, *ia* and *xbar* blockwise, and replicated *ja* and *xsol*. The computation proceeds in time steps, where in each time step, a relaxation subroutine is called five times. This routine produces new values in the local solution vector *xbar*. At the end of each time step, the local solution is committed to the global solution vector, and convergence is

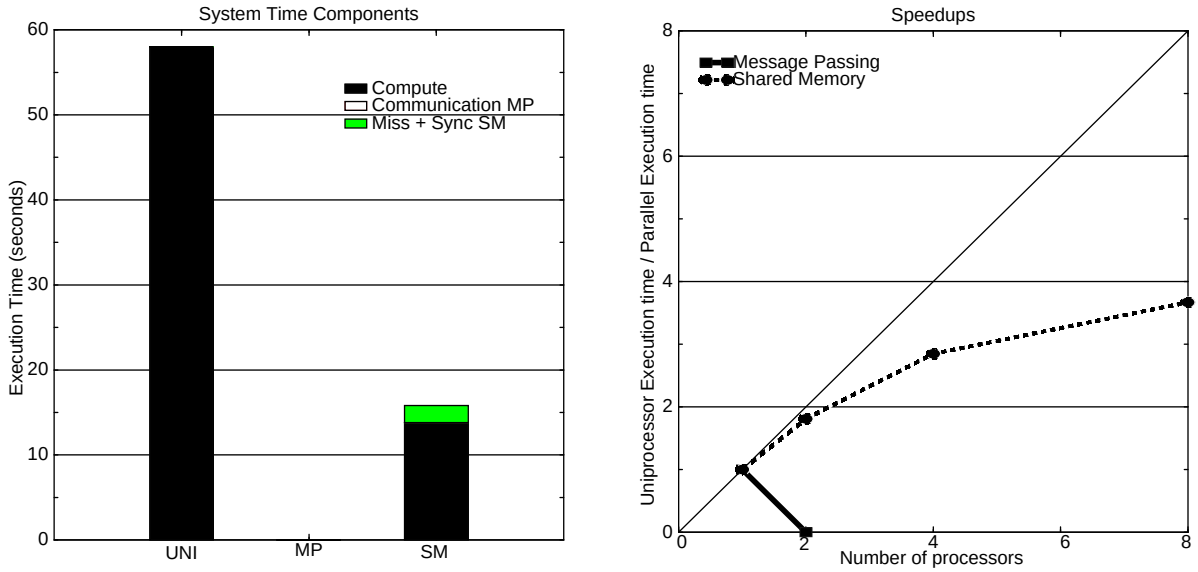

Figure 7: Performance data for *LCP*. We could not run *LCP* for more than 2 nodes.

tested. The communication arises in updating the global solution vector, and in testing for convergence, which entails a reduction. Figure 7 presents the performance data.

Even though the arrays *a* and *ia* are read-only data, the relevant values need to reach each node once: this is an ideal case for the inspector-executor paradigm, but the current *pghpf* compiler resorts to run-time resolution for message passing. The two node execution time for message passing was about 1000 times slower than the uniprocessor case. We could not complete 8 node runs even in several hours. For the shared memory case, we are able to get some speedup (about 3.7) by the use of the INDEPENDENT directive; the accesses to *a* and *ia* are cached automatically.

6 Discussion

Tempest does not enforce a particular cache block size (unit of coherence), neither does it enforce a particular coherence policy. Although we typically use 128 byte blocks, we could use larger blocks to provide us the associated benefits of prefetching. Large blocks, however, are not a panacea, as they may hurt performance if they induce false sharing, or otherwise contribute to useless traffic. However, in most of the programs we studied, larger block size of 512 bytes helped improve shared memory performance. Figure 8 presents comparative speedup results (8 nodes) on message passing, shared memory implemented with 128 byte blocks, and shared memory implemented with 512 byte blocks. Since the block size selection is simply a matter of recompiling user-level library code, we are justified in refining our selection of block size for an application if it improves performance. Similarly, although we have not demonstrated update-based protocols in this study, earlier experiments [9] suggest that they may improve performance over invalidation-based protocols.

7 Conclusion

The goal of this study was to compare a fine-grain distributed shared memory system with compiler implemented shared memory on message-passing hardware. Originally, we intended to address programs that have features that present message-passing compilers a continuum of challenge in static analysis. However, we

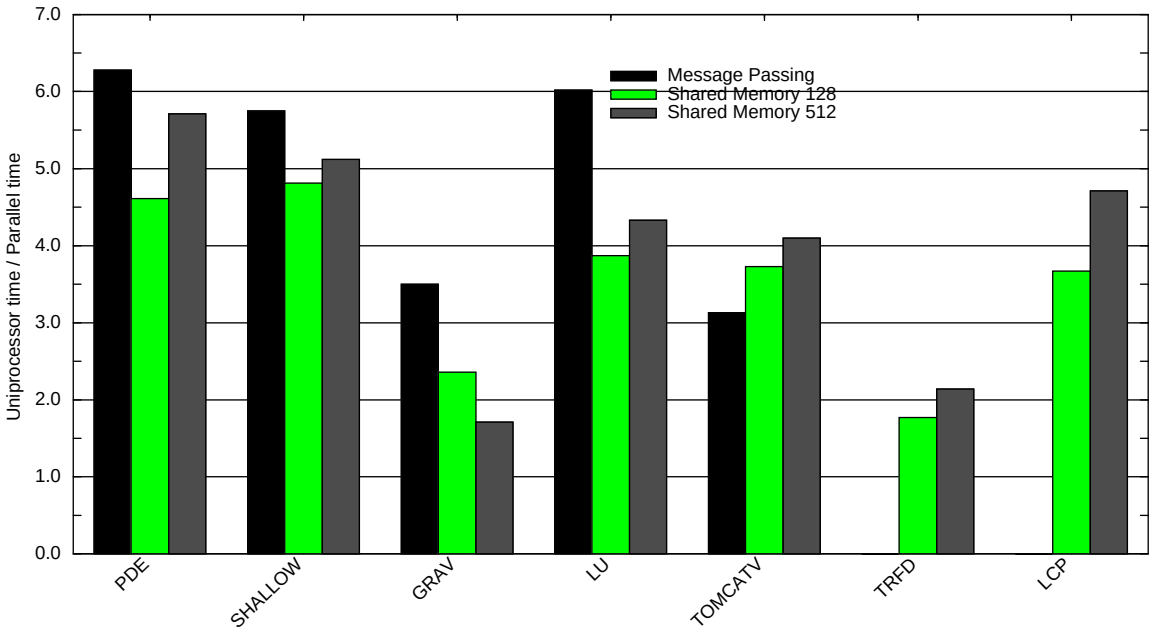

Figure 8: Comparative speedups (8 nodes) on message passing, shared memory with 128 byte cache blocks, and on shared memory with 512 byte cache blocks.

found it very hard to obtain third-party HPF programs that were different from grid-based iterative computations, and were forced to produce two such programs ourselves. Perhaps this bias is due to the fact that the widely available compilers handle only "regular" programs well, and there is little motivation to use HPF for irregular codes. Thus, our current programs reflect an all-or-nothing scenario in terms of applicability of static analysis. In the set of seven programs we studied, five have a regular structure for which compiler analysis for message passing is satisfactory, even though speedups are often not. The remaining two are programs for which such analysis fails. We hope that compiler writers (and application programmers) view fine-grain DSM as a credible means for supporting a wider variety of programs.

In our limited experience, explicit message passing has a substantial advantage (up to 55%) only when an application can express all its communication as coarse-grain shifts and broadcasts. On the other hand, shared memory provides a good way of executing programs for which imperfect analysis prevents a compiler from generating good message-passing code. While more experimentation is undeniably required, this preliminary evidence indicates that compiler writers should consider delegating the synthesis of a shared address space to an underlying system. With the Tempest interface, a compiler can still use efficient message passing when analysis is precise and fall back on shared memory for other cases. Several methods to improve shared memory performance, such as barrier elimination and the use of update based protocols, may bridge the gap in cases in which shared memory trails message passing.

8 Acknowledgments

Yannis Schoinas, Steve Reinhardt and Marc Dionne provided invaluable assistance with using the Blizzard prototype used in this study. Krisna Kunchithapadam, Guhan Vishwanathan and Manish Gupta provided comments on earlier versions of this paper. Finally, Portland Group, Inc. provided us with their HPF compiler infrastructure, as well as prompt answers to many questions about the source code.

References

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and Computation Transformations for Multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, July 1995.

[3] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 112–125, June 1993.

[4] William Blume and Rudolf Eigemann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[7] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[8] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global Communication Analysis and Optimization. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[9] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 61–75, October 1994.

[10] Michal Cierniak and Wei Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[11] G. Cybenko, J. Bruner, S. Ho, and S. Sharma. Parallel Computing and the Perfect Benchmarks. Technical Report 1191, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, November 1991.

[12] Ian Foster. Task Parallelism and High Performance Languages. *IEEE Parallel and Distributed Technology: Systems and Applications*, 2(3):?–?, Fall 1994.

[13] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessor Systems*. PhD thesis, Rheinischen Friedrich-Wilhelms-Universit"at, 1989.

[14] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[15] Manish Gupta and Prithviraj Banerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[16] Manish Gupta and Edith Schonberg. A Framework for Exploiting Data Availability to Optimize Communication. In *Languages and Compilers for Parallel Computing (Proceedings of the Sixth Internationa Workshop)*, pages 216–233. Springer-Verlag, 1994.

[17] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[18] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical Report 93-214, Department of Computer Science, Rice University, November 1993.

[19] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[20] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *High Performance Fortran Handbook*. MIT Press, Cambridge, Mass., 1994.

[21] D.A. Koufaty, X. Chen, D.K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, page ?, 1995.

[22] James R. Larus. Compiling for Shared-Memory and Message-Passing Computers. *ACM Letters on Programming Languages and Systems*, 2(1–4):165–180, March–December 1994.

[23] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[24] Ravi Mirchandaney, Seema Hiranandani, and Ajay Sethi. Improving the Performance of DSM Systems via Compiler Involvement. In *Proceedings of Supercomputing '94*, pages 763–772, 1994.

[25] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth Proceedings of Symposium on Architectural Support for Programming Languages and Operations Systems*, pages 62–73, October 1992.

[26] D. Padua, R. Eigenmann, J. Hoeflinger, P. Peterson, P. Tu, S. Weatherford, and K. Faigin. Polaris: A New-Generation Parallelizing Compiler for MPP's. Technical Report 1306, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, June 1993.

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

[28] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[29] Anne Marie Rogers. Compiling for Locality of Reference. Technical Report TR 91-1195, Department of Computer Science, Cornell University, March 1991. PhD thesis.

[30] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[31] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[32] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[33] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, 1992.

[34] Chau-Wen Tseng. *An Optimizing FORTRAN D Compiler for Distributed Memory MIMD Machines*. PhD thesis, Rice University, January 1993. Also available as Rice CRPC-TR93291-S.

[35] Chau-Wen Tseng. Compiler Optimization for Eliminating Barrier Synchronization. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 144–155, August 1995.

[36] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennesy. SUIF: An Infrastructure for Reseasrch on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.