Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers

Alan L. Cox†, Sandhya Dwarkadas‡, Honghui Lu† and Willy Zwaenepoel†

† Rice University
Houston, TX 77005-1892
{alc, hhl, willy}@cs.rice.edu

‡ University of Rochester
Rochester, NY14627-0226
sandhya@cs.rochester.edu

Abstract

In this paper, we evaluate the use of software distributed shared memory (DSM) on a message passing machine as the target for a parallelizing compiler. We compare this approach to compiler-generated message passing, hand-coded software DSM, and hand-coded message passing. For this comparison, we use six applications: four that are regular and two that are irregular.

Our results are gathered on an 8-node IBM SP/2 using the TreadMarks software DSM system. We use the APR shared-memory (SPF) compiler to generate the shared memory programs, and the APR XHPF compiler to generate message passing programs. The hand-coded message passing programs run with the IBM PVMe optimized message passing library. On the regular programs, both the compiler-generated and the hand-coded message passing outperform the SPF/TreadMarks combination: the compiler-generated message passing by 5.5% to 40%, and the hand-coded message passing by 7.5% to 49%. On the irregular programs, the SPF/TreadMarks combination outperforms the compiler-generated message passing by 38% and 89%, and only slightly underperforms the hand-coded message passing, differing by 4.4% and 16%. We also identify the factors that account for the performance differences, estimate their relative importance, and describe methods to improve the performance.

1. Introduction

This paper evaluates the potential for using software distributed shared memory (DSM) [2, 6, 11] as a target for a parallelizing compiler on a message passing machine. We compare this approach with the more common method whereby the compiler directly targets the underlying message passing system (e.g., [9]).

Shared memory is an attractive target, especially for irregular applications, because the DSM system greatly eases the burden on the parallelizing compiler. Compilers generating message passing code for irregular accesses are either inefficient or quite complex (e.g., the inspector-executor model [15]). Without the inspector-executor model, imprecise compiler analysis leads to large amounts of communication in the compiler-generated message passing programs. Because the compiler does not know what data will be accessed, it broadcasts all data in each processor's partition to all other processors. Compiling to a DSM system avoids this penalty because the DSM system provides on-demand data communication and automatic data caching. The addition of an inspector-executor adds a lot of complexity to the message-passing compiler. The combination of a shared memory compiler and a DSM system avoids this complexity. The goal of this paper is to evaluate the efficiency of this combination of a shared memory parallelizing compiler and a software DSM system, both for regular and irregular applications. We include regular applications in our application suite, because we want to investigate the general applicability of our approach.

Our experimental environment is an 8-node IBM SP/2, on which we use the TreadMarks DSM system [2] to provide shared memory. We use 6 applications: Jacobi, Shallow, Modified Gramm-Schmidt (MGS), 3-D FFT, IGrid, and Non-Bonded Force (NBF). The first four have regular access patterns, while the latter two are irregular. We use the APR Forge XHPF compiler [3] to generate message passing code, and the APR Forge SPF compiler [4] to generate shared memory code. We present the performance of the compiler-generated message passing and shared memory programs. In addition, we present the performance of hand-coded message passing and shared memory programs for the same applications.

For the regular applications, the compiler-generated message passing programs outperform the compiler-generated shared memory programs by 5.5% to 40%. We identify two groups of causes contributing to these differences. First, there are the known factors contributing to the performance differences between message passing and software DSM, regardless of whether the programs are compiler-generated or hand-coded [13]. For the applications in this study, the relevant factors are the overhead of the shared memory implementation (detecting modifications), the separation of data and synchronization, the absence of data communication aggregation in shared memory, and false sharing. Second, there are, for some programs, substantial differences between the compiler-generated and hand-coded shared memory programs. Reasons include redundant synchronization and lack of locality between successive parallel loops or between parallel loops and sequential code. To determine the relative importance of each factor that separates compiler-generated shared memory from hand-coded message passing, we hand-modify the compiler-generated shared memory code to eliminate one factor at a time.

For the irregular programs, the situation is reversed. The compiler-generated shared memory programs outperform the compiler-generated message passing programs by 38% and 89%. More importantly, the compiler-generated shared memory programs achieve performance close to that of the hand-coded message passing programs, differing by 4.4% to 16%. Since the hand-coded message passing programs provide a reasonable upper bound on the performance achievable, we believe that at least for this environment and these applications, more sophisticated compilers targeting message passing could not substantially outperform the compiler-generated shared memory programs.

The outline of the rest of this paper is as follows. In Section 2 we discuss the compilers and the DSM run-time system that we used, and the interface between them. In Section 3 we describe the experimental environment and the methodology used. Section 4 provides a brief overview of the applications. Performance results are presented next, in Section 5 for the regular applications and in Section 6 for the irregular applications. Section 7 summarizes the results. Section 8 proposes further optimizations. We conclude in Section 9.

2. Compilers, Run-Time, and Interface

A modified version of the Forge SPF source-to-source compiler is used to generate shared memory programs for TreadMarks. This section gives a brief overview of the Forge SPF translator, the TreadMarks DSM system, and the interface between the two. We also present the relevant features of the Forge XHPF compiler.

2.1. Forge SPF

Forge SPF is a parallelizing Fortran compiler for shared memory multiprocessor architectures. SPF analyzes a Fortran 77 program annotated with loop parallelization directives, and produces a version in which the DO loops are parallelized with explicit calls to APR's POSIX threads-based parallel run-time routines. The loop parallelization directives specify which loops to parallelize and how to partition the loop iterations. The run-time routines follow a fork-join model of parallel execution, in which a single master processor executes the sequential portion of the program, assigning computation to worker processors when a parallel loop is encountered.

The SPF compiler uses a simple block or cyclic loop distribution mechanism. Reduction on scalar variables is implemented by allocating the reduction variable in shared memory, and declaring a private copy of the reduction variable. Each processor first reduces its own data to the local copy of the reduction variable. At the end of the parallel loop, each processor acquires a lock, and modifies the shared reduction variable.

The compiler allocates in shared memory all the scalars or arrays that are accessed in parallel loops, regardless of whether two processors will access the same storage location or not. Shared arrays are padded to page boundaries in order to reduce false sharing.

2.2. TreadMarks

TreadMarks [2] is a user-level software DSM system that runs on most Unix platforms. It provides a global shared address space on top of physically distributed memory. The parallel processors synchronize via primitives similar to those used in hardware shared memory machines: barriers and *mutex* locks. In Fortran, the shared data are placed in a common block loaded in a standard location. TreadMarks also provides functions to start up and terminate processors, query the number of processors, and query the processor id.

TreadMarks relies on user-level memory management techniques provided by the operating system to detect accesses to shared memory at the granularity of a page. A *lazy invalidate* version of *release consistency* (RC) and a multiple-writer protocol are employed to reduce the amount of communication involved in implementing the shared memory abstraction.

RC is a relaxed memory consistency model. In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category divided into *acquire* and *release* accesses. RC requires ordinary shared memory updates by a processor p to become visible to another processor q only when a subsequent release by p becomes visible to q via some chain of synchronization events. In practice, this model allows a processor to buffer multiple writes to shared data in its local memory until a synchronization point is reached.

With the multiple-writer protocol, two or more processors can simultaneously modify their own copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing. The merge is accomplished through the use of *diffs*. A diff is a runlength encoding of the modifications made to a page, generated by comparing the page to a copy saved prior to the modifications.

The *lazy* implementation delays the propagation of consistency information until the time of an acquire. Furthermore, the releaser notifies the acquiring processor of which pages have been modified, causing the acquiring processor to *invalidate* its local copies of these pages. A processor incurs a page fault on the first access to an invalidated page, and obtains diffs for that page from previous releasers.

Barriers have a centralized manager. At barrier arrival, each processor sends a release message to the manager, waits until a barrier departure message is received from the manager, and then leaves the barrier. The manager collects release messages from the processors when it arrives at barrier arrival. After all processors have arrived at the same barrier, the manager broadcasts the barrier departure message to all processors. The number of messages sent in a barrier is $2 \times (n-1)$, where n is the number of processors.

Each lock has a statically assigned manager. The manager records which processor has most recently requested the lock. All lock acquire requests are directed to the manager, and, if necessary, forwarded to the processor that last requested the lock. A lock release does not cause any communication.

2.3. An Improved Compiler-Run-Time Interface

In our initial implementation, the fork-join semantics expected by the SPF compiler (see Section 2.1) were implemented in terms of the existing TreadMarks interface, using barriers for synchronization and page faults for communicating control information between the master and the worker processors. After startup, worker processors wait at a barrier for parallel work, while the master processor executes any sequential parts in the program. The master processor calls the barrier to wake up the worker processors before a parallel loop. After the parallel loop, the worker processors wait for the next parallel loop at the same barrier. When all have arrived, the master executes any sequential code that may follow the parallel loop, and wakes up the workers at the next parallel loop.

Each parallel loop is encapsulated by SPF into a new subroutine. Before executing the barrier starting the execution of a parallel loop, the master writes the subroutine to be executed, and the parameters to the subroutine, into locations in shared memory. When the worker processors depart from the barrier, they access these locations in shared memory, and jump to the appropriate subroutine with the arguments provided.

Part of the overhead of this implementation of the fork-join operation comes from the barriers encapsulating a parallel loop. Another part of the overhead is due to the propagation of the loop control variables. The barrier provides stronger synchronization than is necessary in the fork-join model. A barrier is an all-to-all synchronization, of which the arrival is an all-to-one synchronization, and the departure is a one-to-all synchronization. On the other hand, the fork-join model incurs a one-to-all synchronization at the fork, and an all-to-one synchronization at the join. The loop control variables include the loop index for dispatching the subroutine encapsulating the loop, and the subroutine parameters. The two sets of control variables reside in different shared pages, incurring two requests to obtain them for each parallel loop. In summary, for each parallel loop execution, the two barriers require $4 \times (n-1)$ messages, and access to the control variables causes two page faults on each worker, and $4 \times (n-1)$ messages.

To reduce the fork-join overhead, we added all-to-one and one-to-all synchronization to the TreadMarks interface, namely the barrier arrival and the barrier departure. A barrier departure is called before a parallel loop, and a barrier arrival is called after a parallel loop. In addition, the barrier departure carries the loop control variables, avoiding the cost of page faulting them in. This optimization reduces the number of messages from $8 \times (n-1)$ to $2 \times (n-1)$, and has a significant effect on execution time. All results in this paper are obtained with this improved interface.

2.4. Forge XHPF

Forge XHPF is a parallelizing compiler for High Performance Fortran [3] on distributed memory multiprocessor systems. It transforms a sequential Fortran program annotated with subset HPF data decomposition directives and Fortran 90 array syntax into a SPMD (Single Program Multiple Data) parallelized Fortran 77 program. In the SPMD model, the sequential part is executed by all the processors, while the DO loops are distributed across processors. Part of the sequential code is guarded by if statements because only some processors must execute them. The user is required to insert data partitioning directives to specify the distribution of data across the processors. The compiler uses these directives as a seed for parallelizing loops and distributing arrays.

The translated code relies on a small run-time system to handle process creation, to assign loops to processors, and to perform the underlying communication. To implement loop distribution, the run-time system maintains descriptors for all distributed arrays, and tries to generate loop distributions that satisfy the owner-computes rule [16]. In case the communication pattern is unknown at compile time, the compiler inserts instructions to broadcast all the data in a processor's partition (specified by the user) at the end of the parallel loop, regardless of whether the data will actually be used.

3. Environment and Methodology

Our experimental environment is an 8-processor IBM SP/2 running AIX version 3.2.5. Each processor is a thin node with 64 KBytes of data cache and 128 Mbytes of main memory. Interprocessor communication is accomplished over the IBM SP/2 high-performance two-level cross-bar switch. Unless indicated otherwise, all results are for 8-processor runs. We use version 0.10.1 of TreadMarks, with the optimized interface discussed in Section 2.3. We use version 2.0 of the SPF compiler and the the XHPF compiler. Both TreadMarks and the XHPF compiler use the user-level MPL communication library as the underlying message passing system. The hand-coded versions of the message passing programs use PVMe, an implementation of PVM [8] optimized for the IBM SP/2.

Our primary goal is to assess the performance of the compiler-generated shared memory programs. Performance is quantified primarily by speedup, but we also provide statistics on the number of messages and the amount of data exchanged during execution. In addition, we provide the same performance figures for XHPF-generated message passing, hand-coded PVMe message passing, and hand-coded TreadMarks shared memory implementations of the same applications. The XHPF performance numbers provide an indication of the capabilities of current commercial compilers targeting message passing. Comparing SPF-generated shared memory to hand-coded shared memory quantifies the performance loss as a result of compiler generation. Comparing hand-coded shared memory to hand-coded message passing quantifies the performance loss as a result of using the shared memory system. The hand-coded message passing provides an reasonable upper bound on the performance achievable for the program on this platform.

In addition, to quantify the contribution of some of the sources of overhead, we have hand-modified the SPF-generated code to eliminate those sources. We use the enhanced TreadMarks interface proposed by Dwarkadas et al. [7] to achieve communication aggregation, consistency elimination, and pushing (instead of pulling) data to the processors that will use it. We also eliminate redundant barriers in the program.

4. Applications

We use 6 applications (Jacobi, Shallow, MGS, 3-D FFT, IGrid, and NBF). Jacobi, Shallow, MGS, and 3-D FFT all have regular access patterns, while IGrid and NBF are irregular. Jacobi is an iterative method for solving partial differential equations. Shallow is the shallow water benchmark from the National Center for Atmospheric Research. MGS implements a Modified Gramm-Schmidt algorithm for computing an orthonormal basis. 3-D FFT, from the NAS [5] benchmark suite, numerically solves a partial differential equation using three dimensional forward and inverse FFT's. IGrid is a 9-point stencil computation in which the neighbor elements are accessed indirectly through a mapping established at run-time. The NBF (Non-Bonded Force) program is the kernel of a molecular dynamics simulation.

Table 1 presents the data set size as well as the sequential execution time for each application. Sequential execution times are obtained by removing all synchronization from the TreadMarks programs and executing them on a single processor.

Program	Problem Size	Time (sec.)
Jacobi	2048×2048, 100 iterations	72.9
Shallow	1024×1024, 50 iterations	64.7
MGS	1024×1024	56.4
3-D FFT	$128 \times 128 \times 64$, 5 iterations	37.7
IGrid	500×500, 20 iterations	42.6
NBF	32K molecules, 20 iterations	63.9

Table 1. Data Set Sizes and Sequential Execution Time of Applications

5. Results for Regular Applications

Figure 1 presents the 8-processor speedups for each of the regular applications for SPF-generated TreadMarks, hand-coded TreadMarks, XHPF-generated message passing, and hand-coded message passing. The 8-processor speedups are calculated based on the sequential times listed in Table 1. Table 2 provides the corresponding number of messages and amount of data exchanged during the execution. In the following we analyze the differences between SPF-generated shared memory and hand-coded shared memory, the differences between hand-coded shared memory and hand-coded message passing, and present the results of hand-applied optimizations.

5.1. Jacobi

Jacobi is an iterative method for solving partial differential equations. The algorithm employs two arrays - one

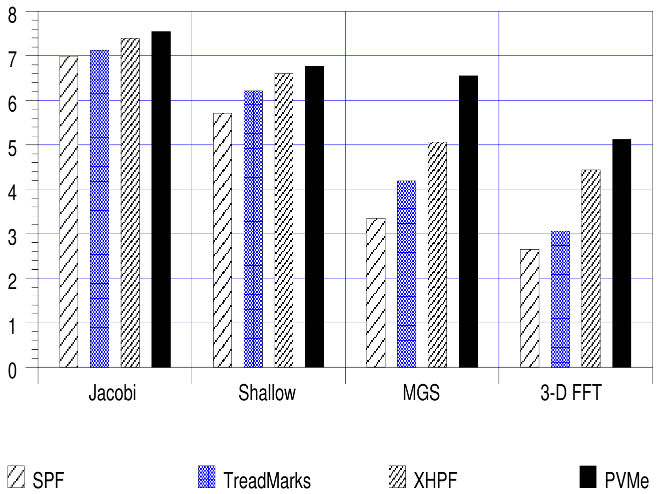

Figure 1. 8-Processor Speedups for Regular Applications for SPF-generated TreadMarks, hand-coded TreadMarks, XHPF-generated message passing and hand-coded PVMe

	Program	SPF	Tmk	XHPF	PVMe
Message Total	Jacobi	8538	8407	4207	1400
	Shallow	13034	11767	7792	1985
	MGS	57283	30457	38905	7168
	3-D FFT	52818	36477	33913	1155
Data Total	Jacobi	989	862	11458	11469
	Shallow	10814	10400	18407	7328
	MGS	59724	55681	29430	29360
	3-D FFT	103228	74107	102763	73401

Table 2. 8-Processor Message Totals and Data Totals (Kilobytes) for Regular Applications for SPF-generated TreadMarks, hand-coded TreadMarks, XHPF-generated message passing and hand-coded PVMe

is the data array and one is the scratch array. There are two phases in each iteration. First, each element is updated according to a four point stencil and the new values are stored in the scratch array, then the scratch array is copied to the data array. Both loops are parallelized. The first loop requires nearest neighbor communication, in which neighboring processors exchange their boundary columns in the data array. The shared memory versions place the data array in shared memory. However, the SPF-compiler also allocates the scratch array in shared memory, because it is accessed in a parallel loop.

The data array is initialized with ones on the edges and zeroes in the interior. The program iterates 101 times on a 2048×2048 real array, in which the last 100 iterations are timed. The speedups of the compiler-generated version for shared memory and message passing are 6.99 and 7.39, respectively, closely approximate those of the hand-coded versions, which are 7.13 and 7.55, respectively.

SPF/Tmk vs. Tmk The 2% difference between the SPF-generated and the hand-coded shared memory programs is due to the use of private memory for the scratch array in the TreadMarks version, and the use of shared memory in the SPF-generated TreadMarks program.

Tmk vs. PVMe Four factors give the message passing programs a slight edge. First, it takes two access faults and four messages in TreadMarks to obtain the new values in a neighboring column. In the message passing program, in contrast, the boundary column is sent directly to its destination in a single message. Second, the shared memory version has separate data (access misses) and synchronization messages, whereas the message passing version sends a single message for both purposes. Third, in the message passing program, no communication is incurred between the first and the second phase of each iteration. In the shared memory model, a barrier has to be inserted between the two phases to make sure the data are not written before they are read (to respect the anti-dependence). Finally, the shared memory programs incur the overhead of detecting modifications to shared memory (twinning, diffing, and page faults). The shared memory program sends much less data than the message passing program, because the modifications are propagated from the edges to the center of the array, and only modified data are sent in TreadMarks.

Results of Hand Optimizations The main overhead in the shared memory programs stems from the first factor, the absence of data aggregation. A hand-modified version of the shared memory program that includes data aggregation achieves a speedup of 7.23 for the SPF-generated version, compared to 7.55 for hand-coded message passing.

5.2. Shallow

Shallow is the shallow water benchmark from the National Center for Atmospheric Research. It solves differential equations using 13 equal-sized two-dimensional arrays in wrap-around format. There are three steps in each iteration, each of which consists of a main loop that updates three to four arrays according to values in some other arrays. Wrap-around copying is applied to the modified arrays after the main loop in each step. The wrap-around copying includes two separate loops to copy the boundary rows and the boundary columns. The parallel versions partition the arrays by column, and require nearest neighbor communication similar to that in Jacobi. In the shared memory version, all of the arrays are shared, and the main loop and the edge row copying are parallelized. The edge column copying is executed sequentially, because the arrays are laid out in column major order. In the message passing version, the edge column copying is executed on the processor at which the data resides (the owner-computes rule).

We use a 1024×1024 grid in our test, where each grid element is represented by a real number. The program runs for 51 iterations, of which the last 50 iterations are timed. At 8 processors, the compiler-generated shared memory and message passing achieve speedups of 5.71 and 6.60, respectively. The hand-coded shared memory and message passing programs achieve speedups of 6.21 and 6.77, respectively.

SPF/Tmk vs. Tmk The slowdown in the SPF-generated shared memory version is caused by redundant synchronization and extra communication. Redundant synchronization occurs because the SPF compiler inserts a pair of synchronizations around every parallel loop, some of which are unnecessary in a hand-coded shared memory program. The extra communication is due to the fact that the sequential part is always executed by the master processor in the fork-join model, regardless of the owner-computes rule.

Tmk vs. PVMe The reasons for the difference between the shared memory programs and the two message passing programs are essentially the same as for Jacobi, namely separation of data and synchronization, absence of data aggregation, and the overhead of the shared memory implementation.

Results of Hand Optimizations By hand, we merged the loops in the SPF-generated program and put in data aggregation. This resulted in an increase in speedup to 5.96, compared to 6.21 for hand-coded shared memory. The remaining difference in speedup is due to the extra communication resulting from the execution of the sequential parts of the code by the master processor.

5.3. MGS

Modified Gramm-Schmidt (MGS) computes an orthonormal basis for a set of N-dimensional vectors. At each iteration i, the algorithm first sequentially normalizes the ith vector, then makes all vectors $j > i$ orthogonal to vector i in parallel. The vectors are divided over the processors in a cyclic manner to balance the load in each iteration. All processors synchronize at the end of an iteration.

On 8 processors, the speedup achieved by the SPF-generated TreadMarks version is 3.35, XHPF 5.06, the hand-coded TreadMarks version achieves 4.19, and PVMe 6.55. The difference between the XHPF version and the hand-coded message passing version is due to the SPMD model used by XHPF, in which all processors participate in normalization of the ith vector.

SPF/Tmk vs. Tmk The difference between the hand-coded TreadMarks version and the SPF-generated one is also a result of extra communication in the SPF-generated version. Since normalization of the ith vector during iteration i is part of the sequential code, it is always done on the master processor, requiring the vector to move from its assigned processor to the master. The hand-coded version is written such that the normalization happens on the processor the vector is assigned to.

Tmk vs. PVMe The difference between the message passing and the shared memory programs results primarily from the two message passing programs' ability to broadcast the ith vector on the ith iteration. In contrast, in the shared memory programs, this vector needs to be paged in by all other processors. Furthermore, the separation of synchronization (barrier) and data (ith vector) leads to additional communication.

Results of Hand Optimizations We hand-modified the program to merge the data and the synchronization, and modified TreadMarks to use a broadcast. The speedup improved to 5.09 from 4.19.

5.4. 3-D FFT

3-D FFT, from the NAS [5] benchmark suite, numerically solves a partial differential equation using three dimensional forward and inverse FFT's. Assume the input array A is of size $n_1 \times n_2 \times n_3$, organized in column-major order. The 3-D FFT first performs a n_1-point 1-D FFT on each of the $n_2 \times n_3$ complex vectors. Then it performs a n_2-point 1-D FFT on each of the $n_1 \times n_3$ vectors. Finally, it performs a n_3-point 1-D FFT on each of the $n_1 \times n_2$ vectors. The complex array is reinitialized at the beginning of each iteration. After that, an inverse 3-D FFT is applied to the complex array. The resulting array is normalized by multiplying each element in the array by a constant. Finally, the checksum is computed by summing 1024 elements in the array.

There are six parallel loops in each iteration, one for initializing the array, three for the FFT in each of the three dimensions, one for the normalization, and one for the checksum. Each iteration starts with a block partition on n_3, and keeps using this partition until the beginning of the n_3-point FFT. The n_3-point FFT works on a different data partition (block partition on n_2), thus requiring a transpose. The new data partition is used through the end of the iteration. The TreadMarks program has two barriers, one after the transpose, and another one after the checksum is computed.

We ran the program on a $128 \times 128 \times 64$ double precision, complex array for 6 iterations, excluding the first iteration from the measurement. At 8 processors, the compiler-generated programs achieve a speedup of 2.65 for the shared memory version and 4.44 for the message passing version. The hand-coded shared memory program and message passing programs achieve speedups of 3.06 and 5.12, respectively.

SPF/Tmk vs. Tmk The SPF-generated shared memory program incurs additional overhead by inserting synchronization around each parallel loop.

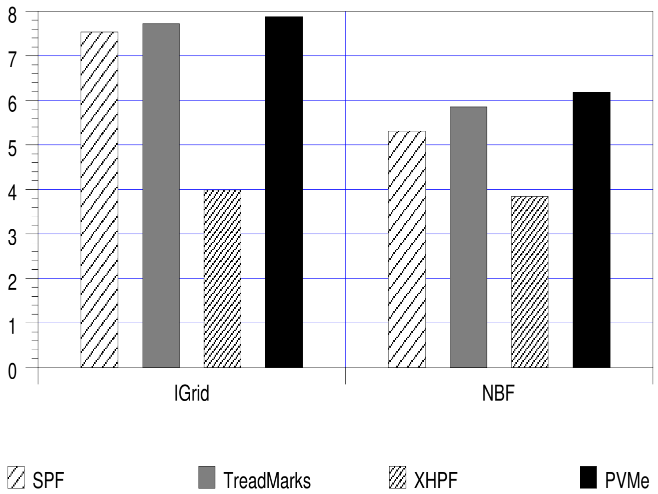

Figure 2. 8-Processor Speedups for Irregular Applications for SPF-generated TreadMarks, hand-coded TreadMarks, XHPF-generated message passing and hand-coded PVMe

Tmk vs. PVMe The sizable difference in performance between message passing and shared memory occurs during the transpose. Because of the huge data size, and because of the fact that the shared memory versions fault in the data one page at a time, the number of messages in the shared memory version is about 30 times higher than in the hand-coded message passing version.

Results of Hand Optimizations In a hand-modified version of the SPF-generated program that includes data aggregation, speedup rose to 5.05, very close to the 5.12 observed for the hand-coded message passing. Without changing the program drastically, we are able to merge only the first two 1-D FFT loops. This optimization has little effect.

6. Results for Irregular Applications

Figure 2 presents the 8-processor speedups for each of the irregular applications for SPF-generated TreadMarks, hand-coded TreadMarks, XHPF-generated message passing, and hand-coded message passing. The 8-processor speedups are calculated using the sequential times listed in Table 1. Table 3 provides the corresponding number of messages and amount of data exchanged during the execution.

6.1. IGrid

IGrid is a relaxation code that utilizes a nine point stencil. The neighbor elements are accessed indirectly through a mapping established at run-time. The program works on two arrays, one for keeping the newly computed data in each step, and one for storing the old data from the previous step. At the beginning of the program, the old array is initialized to all ones, then two spikes are added to the middle and the lower right corner of the array. During each step, the new value of each grid point is computed according to the old values of its eight neighbors and itself. The new array and the old array are switched at the end of each step. At the end of the program, the code finds the maximum and minimum values in a 40×40 square in the middle of the array, and computes the sum of the values in the square.

	Program	SPF	Tmk	XHPF	PVMe
Message Total	IGrid	3806	1246	34769	320
	NBF	14836	13194	45895	960
Data Total	IGrid	7374	131	140001	640
	NBF	1543	228	163775	31457

Table 3. 8-Processor Message Totals and Data Totals (Kilobytes) for Irregular Applications for SPF-generated TreadMarks, hand-coded TreadMarks, XHPF-generated message passing and hand-coded PVMe

Due to the indirection array, the compilers cannot recognize the data access pattern in the program. Both of the compilers are instructed to parallelize the main computation loop assuming the iterations of the loop are independent. The shared memory compiler partitions the iterations among processors and encapsulates the parallel loop in a pair of synchronization operations. Although the message passing compiler is told to partition the grid into blocks, since the compiler does not know what data will be needed during the next step, it makes each processor broadcast its whole block at the end of each step. The max-min finding and checksum computation are recognized as reductions.

We use two 500×500 single precision matrices in our experiments. Of the 20 iterations executed, the last 19 iterations are measured in order to avoid startup effects. At 8 processors, the speedup for SPF-generated TreadMarks is 7.54, compared to 7.88 for hand-coded message passing. The speedup of the XHPF-generated version is only 3.85. The large number of messages and the large amount of data exchanged (see Table 3) explain the drop in performance for the XHPF program. The shared memory versions fetch data on-demand, and the run-time system automatically caches previously accessed shared data. Hence, only the data that is actually modified remotely and accessed is communicated.

6.2. NBF

NBF is the kernel of a molecular dynamics simulation. The program simulates the behavior of a number of molecules. Each molecule has a list of “partners”, molecules that are close enough to it to exert a non-negligible effect on the molecule. For each molecule, the program goes through the list of partners, and updates the forces on both of them based on the distance between them. At the end of each iteration, the coordinates of the molecules are updated according to the force acting on them. The program is parallelized by block-partitioning the molecules among processors. Each processor accumulates the force updates in a local buffer, and adds the buffers together after the force computation loop.

The XHPF compiler cannot recognize the data access pattern due to the indirection array. It therefore makes each processor broadcast its local force buffer, and the coordinates of all its molecules. The SPF compiler inserts a synchronization statement at the end of an iteration. At run-time, this synchronization causes TreadMarks to invalidate the modified pages. Individual processors then take page faults on those pages they access. Since this is typically only a small subsection of the array, the number of messages and the amount of data is far smaller than for the XHPF-generated message passing program.

The differing amounts of communication are clearly reflected in Table 3, and in the speedups for the different versions. In decreasing order, PVMe achieves a speedup of 6.18, hand-coded TreadMarks 5.86, SPF-generated TreadMarks 5.31, and finally XHPF-generated message passing 3.85.

7. Summary of Results

On the regular programs, both the compiler-generated and the hand-coded message passing outperform the SPF/TreadMarks combination: the compiler-generated message passing by 5.5% to 40%, depending on the program, and the hand-coded message passing by 7.5% to 49%. In general, three factors favor the message passing programs: better data aggregation, combined synchronization and data transfer, and no overhead for shared memory coherence.

On the irregular programs, the SPF/TreadMarks combination outperforms the compiler-generated message passing by 38% and 89%. Performance approaches that of the hand-coded message passing, differing by 4.4% and 16%. The small difference between SPF/TreadMarks and hand-coded message passing suggests that, at least for this environment and these applications, more sophisticated compilers targeting message passing could not substantially outperform the compiler-generated shared memory programs. Compared to the compiler-generated message passing, the SPF/TreadMarks combination benefits from on-demand fetching of data, as well as caching of previously accessed data by the run-time system.

On both the regular and the irregular programs, the hand-coded TreadMarks outperforms the SPF/TreadMarks combination. The difference varies from 2% to 20%. In general, two factors account for the difference. The compiler-generated shared-memory programs have excess synchronization and additional data communication. The latter is because there is less processor locality in the programs' data access patterns.

Keleher and Tseng [10] perform a similar study which also compares the performance of compiler-generated DSM programs with compiler-generated message passing programs. Instead of using commercial Fortran compilers to compile all the programs, they use the Stanford SUIF [1] parallelizing compiler version 1.0 to generate parallel C programs for the DSM system, and the commercial IBM HPF or DEC HPF compilers to generate the parallel Fortran programs in message passing. Using a different set of applications, they arrive at results similar to ours.

8. Further Optimizations

In Section 5, we have shown the considerable benefits of hand-applied optimizations for the SPF-generated DSM programs. The optimizations include aggregating data communication, merging synchronization and data, and pushing data instead of the default request-response data communication in the DSM system. Dwarkadas et al. [7] have shown that these optimizations can be implemented automatically by a compiler and DSM runtime system. Those techniques could be integrated with the APR compiler.

Elimination of redundant barriers was proposed by Tseng [17] in the context of automatic parallelization for hardware distributed shared memory machine. His results show a significant reduction in the number of barriers, although only a limited reduction in the execution time. We manually applied this optimization to the applications in this study, and showed that with the high cost of synchronization in this environment, improvements can be substantial.

Additionally, our results indicate the need to optimize the SPF-generated applications for locality of access, similar to the optimizations applied in the XHPF compiler. These optimizations will improve the performance not only of software DSM systems but also of hardware shared memory systems.

We plan to further explore the benefits of customizing DSM systems for compiler-generated shared memory programs, and expect more gains in performance when scaling to a large number of processors. These enhancements will include efficient support for reductions, more aggressive methods of eliminating consistency overhead based on synchronization and access pattern information, and dynamic load balancing support.

Our results indicate that with minimal compiler support, our software DSM system has performance comparable to hand-coded message passing for the irregular applications we have considered. The inspector-executor model [15] has been proposed to efficiently execute irregular computations in the message passing paradigm. Mukherjee et al. [14] compared the CHAOS inspector-executor system to the TSM (transparent shared memory) and the XSM (extendible shared memory) systems. They concluded that TSM is not competitive with CHAOS, while XSM achieves performance comparable to CHAOS after introducing several hand-coded special-purpose protocols. In a more recent paper [12], we compared CHAOS to TreadMarks with simple compiler support for describing accesses to the indirection array. With the compiler support, the TreadMarks DSM system achieves similar performance to the inspector-executor method supported by the Chaos run-time library. The same compiler support can also be used for automatically generated DSM programs.

9. Conclusions

In this paper, we evaluate the efficiency of combining a parallelizing compiler and a software DSM system, both for regular and irregular applications. The results show that for regular applications, the compiler-generated message passing programs outperform the compiler-generated DSM programs by 5.5% to 40%, while for the irregular applications, the compiler-generated DSM programs outperform the compiler-generated message passing programs by 38% and 89%, and underperform the hand-coded message passing programs only by 4.4% and 16%.

This study shows that software DSM is a promising target for parallelizing irregular applications. With appropriate enhancements to the compiler and DSM system, we have also shown that the performance of regular applications can match that of their message passing counterparts, thus making software DSM a general parallelizing platform for all applications.

10. Acknowledgments

We thank John Levesque, Gene Wagenbreth and Allan Jacobs of Applied Parallel Research, Inc. (APR) for providing us with, and modifying their compilers to generate code for TreadMarks. This work is supported in part by the National Science Foundation under Grants CCR-9410457, BIR-9408503, CCR-9457770, CCR-9502500, CCR-9521735, CDA-9502791, and MIP-9521386, by the Texas TATP program under Grant 003604-017, and by grants from IBM Corporation and from Tech-Sym, Inc.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[2] C. Amza, A. 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, Feb. 1996.

[3] Applied Parallel Research, Inc. *FORGE High Performance Fortran User's Guide*, version 2.0 edition.

[4] Applied Parallel Research, Inc. *FORGE Shared Memory Parallelizer User's Guide*, version 2.0 edition.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[6] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[7] S. Dwarkadas, A. 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*, Oct. 1996.

[8] G. Geist and V. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, pages 293–311, June 1992.

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

[10] P. Keleher and C. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[12] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Software distributed shared memory support for irregular applications. 1996. Submitted for publication.

[13] H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings SuperComputing '95*, Dec. 1995.

[14] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed memory machines. In *Proceedings of the 5th Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[15] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. *Concurrency:Practice and Experience*, 3(6):573–592, Dec. 1991.

[16] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Houston, Jan. 1993.

[17] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th Symposium on the Principles and Practice of Parallel Programming*, July 1995.