

An Evaluation of State Sharing Techniques in Distributed Operating Systems *

Ranjit John

R. Ananthanarayanan

Mustaque Ahamad

Umakishore Ramachandran

Silicon Graphics Inc.
Mountain View, CA

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Ajay Mohindra

IBM T. J. Watson Research Center
Yorktown Heights, NY

Abstract

A shared memory abstraction in distributed systems (DSM) provides ease of programming but could be costly to implement. Many protocols have been proposed recently that are based on different approaches for exploiting program semantics. We have implemented four different protocols that embody the different memory semantics and have evaluated them using applications that capture a wide range of state sharing patterns. Our main goal is to quantify the relative performance of these protocols with respect to representative applications. One of the results is that memory systems that provide weaker consistency or provide consistency only at well defined synchronization points (e.g., release consistency), cannot completely avoid the overhead associated with DSM. The disparity between the performance using these protocols and programmer controlled state sharing varied from being insignificant to as high as 45% depending on application characteristics and system size. The use of synchronization information in conjunction with user directives will bridge this performance gap but may decrease the programming ease of DSM.

*This work was supported in part by the National Science Foundation under grants CCR-8619886, CCR-9106627, MIP-9058430 and ARPA contract N00174-93-K-0150.

1 Introduction

The abstractions provided by a distributed operating system for sharing state between processes executing on different nodes have significant impact on the performance and programming of distributed applications. Message passing, which can be supported efficiently in a distributed environment, is a difficult programming paradigm since a process needs to determine *a priori* the part of the state that needs to be shared with another process and then explicitly send/receive it. Remote procedure calls (RPCs) extend the procedure call abstraction and simplify programming but could suffer from certain performance problems, such as the inability to exploit locality of references. For example, when the same call or call to the same process are made repeatedly, communication costs can be avoided by bringing the state of the callee to the caller node and executing the calls locally. In general, such benefits can only be realized if the system provides a shared memory abstraction across nodes¹. The shared memory abstraction in a distributed system is called *distributed shared memory* (DSM). The programming task is simplified by DSM because both local and remote state can be accessed uniformly. Since nodes in a distributed system do not physically share memory, the DSM abstraction has to be implemented in software. Many researchers have investigated [18, 6, 14, 7, 2, 20] distributed shared memories.

Several approaches have been investigated to mitigate the performance problems of DSM systems. These include weaker consistency for shared data [19, 2], use of synchronization information in maintaining coherence [14, 7, 17] and user-directed coherence [6, 3]. These approaches have some impact on programming but it is claimed that they avoid unnecessary communication and hence provide better performance. However, there has been no study to investigate the relative gains made possible in distributed systems by these different approaches.

In this paper, we present experimental evaluation of the different approaches that have been proposed for implementing memory consistency in DSM systems. Our goal is not to investigate how to make an application programmed using DSM competitive in terms of performance with their implementations based on message passing. We focus on quantifying the performance gains made possible by the different approaches for maintaining DSM coherence, with a view to helping an operating systems designer choose a protocol (or a set of protocols) for DSM. We consider four different approaches:

1. The first memory system, M_0 , provides a sequentially consistent memory. Since it provides standard shared memory semantics, it is the easiest to program and also provides a reference point to evaluate the performance of other approaches.

¹Migration can be used to realize locality benefits [13] but cannot be used when more than one node attempts to make the state of the callee local. This would require coherence maintenance of the state and hence would be similar to a shared memory abstraction.

2. The second memory system, M_1 , is based on the weak consistency approach. In particular, it is derived from causal orderings [16, 8], established because of interactions between processors. M_1 implements *causal memory* [2] which guarantees that all processors view memory operations in an order that is consistent with causality. Memory operations that are not causally ordered can be seen by different processors in different order which results in weaker consistency.
3. M_2 , the third memory system we implement makes use of synchronization operations in the coherence activities for shared data. This approach has been advocated by several researchers [11, 1, 14, 7, 17]. In our implementation, we buffer the writes to shared data locally and propagate them to other nodes at certain synchronization points in the program.
4. In our last system, called M_3 , user provided information is used to direct coherence activities. An application programmer provides explicit associations between synchronization variables and corresponding shared data. M_3 makes use of such associations to perform the minimal amount of consistency activity needed for the correct execution of the program, and thus comes close to message passing in terms of the communication requirements [7, 3].

In our study, the first three systems (M_0 to M_2) differ from the last one because they are all page based. In other words, a page is the unit of data consistency in these systems whereas consistency is maintained at the level of application defined data items in M_3 . Furthermore, programming of applications with the M_3 memory system differs from others because programmers need to establish associations between shared data and synchronization variables.

We have implemented DSM systems based on all the four above approaches using a uniform set of mechanisms. A comprehensive evaluation of these DSM systems requires a set of applications that capture a range of data sharing patterns. We have chosen the NAS kernels [4] which represent the access patterns of many scientific computations. In addition, we have implemented SOR, matrix multiplication, and a branch-and-bound solution to the traveling salesperson problem.

The results of our experiments lead to several interesting insights. One of the results is that unless the programmer provides information that is used in coherence maintenance (e.g. annotations in Munin [6] or associations of Midway [7]), DSM protocols will have performance overheads. These overheads manifest as extra messages, the amount of data transferred in these messages and the additional processing introduced by them. Thus, reducing the number of messages is the key to improving performance of a DSM system. This can be seen in the results of our experiments because M_2 , the system based on release consistency, does not provide significantly better performance than M_0 . Although M_2 allows

the sending of consistency related messages to be postponed until a release synchronization operation is reached, such messages are sent to all nodes that cache modified data and hence the number of messages is not reduced. The large page size supported by typical workstations also has a negative impact on the page based systems ($M_0 - M_2$). M_3 , which sends fewer and smaller messages, provides better performance than the other systems considered, since it only sends the new values to the node (or nodes) that next acquires the synchronization variable associated with the updated data. The explicit associations further improve performance by reducing message size because information about all changes to shared data is not propagated. These results show that the ease of programming with a DSM abstraction does have an associated cost irrespective of the DSM consistency protocol when users are not required to provide additional information for consistency maintenance.

Section 2 describes the system we have chosen to implement the different memory systems. We provide further details of these memory systems and the protocols that implement them in Section 3. Section 4 summarizes the applications and their performance is discussed in Section 5. Section 6 presents some observations and concluding remarks are given in Section 7.

2 System

The system used in our research consists of a set of Sun 3/60 workstations connected by a 10Mb/s Ethernet. The protocols for the memory systems M_0 to M_3 are implemented in the Clouds distributed operating system. We use page faults, access violation, and synchronization events to perform coherence activities. For example, when a reference is made to a page that is not cached, the fault handler requests the page from the appropriate data manager (there is a one-to-one correspondence between a given page and a data manager which is similar to the fixed manager concept in [18]). The implementation of the synchronization constructs (locks, semaphores and barriers) is centralized. For a given synchronization variable, a single server maintains its state and the queue of processes blocked on it. We have chosen to implement the coherence protocols in the operating system using the same low-level communication mechanisms providing uniformity and control over events that may interfere with the experiments.

3 Memory Systems

3.1 M_0 – Sequentially Consistent

As stated earlier, M_0 is our base memory system and it provides sequential consistency (SC). We have implemented the invalidation based protocol used in the Ivy system [18] with a fixed

manager. We added several optimizations to the basic protocol. In particular, we control thrashing between nodes by pinning a page [10], and also avoid the re-sending of page data due to double faults [15].

3.2 M_1 – Weaker Consistency

The consistency guarantees provided by M_1 are much weaker than M_0 . It differs from M_0 because it guarantees that processes only agree on the order of those memory operations that are ordered by *causality*. Similar to a message passing system, the causal order we use to relate operations is based on the *program* (local) order at processes and a *reads-from* order that is established between a write and its subsequent reads. M_1 provides weaker consistency than M_0 because values written by concurrent operations can be read by different processes in different orders.

The implementation of M_1 is structured similar to M_0 . One main difference is that invalidations are local to a node and are performed when the node suspects that a read of a cached page may potentially violate the causal orderings. This is done when a page is brought into the memory of a node as a result of a page fault. We use version vectors [12] to detect when a page is overwritten. Version vectors are maintained by each process and a version is associated with each page. Cached pages that have an older version than the values in the version vector received with an incoming page are locally invalidated. Due to weaker consistency, M_1 allows readers to co-exist with a single writer so that readers can continue to read previously cached values because invalidations are only local. Thus, write-read false sharing is handled naturally by M_1 . We handle write-write false sharing by allowing a single node to write a page at any given time. Similar to M_0 , we use pinning to avoid the situation where a page continuously moves between nodes where it is written. When an application is properly synchronized, instead of performing local invalidations each time a page is brought in, it suffices if they are done when an acquire operation on a synchronization variable completes. Also, the version vectors maintained by nodes only need to be sent with synchronization variables (e.g., *locks*).

3.3 M_2 – Exploiting Synchronization

M_2 guarantees sequential consistency only at well defined points in the program. In particular, if the programs are written with some synchronization model in mind, it is possible to defer consistency actions to certain synchronization points. For example, release consistency [11] and buffered consistency [17] allow consistency actions to be deferred until the exit point to a synchronization region, while lazy release consistency [14] allows them to be deferred until the entry point to a synchronization region.

The implementation of M_2 is as follows. A node records all modifications to a page in a

shadow copy (transparent to the program). Prior to exiting a synchronization region (as in release consistency), an XOR of the original page and its shadow is generated for each dirty page (similar to the *diff* in [6]). An optimization that is employed in this implementation is to limit the size of the *diff* messages by recognizing when there is no change in the page beyond a certain offset, and truncating the *diff* message after this offset. The modifications to the data are sent to the appropriate data managers. A data manager merges the modifications to its copy of the page, and also sends them to all nodes that have a copy of the page. Upon propagating the modifications, the program is allowed to exit the synchronization region (e.g., complete an *unlock* or *barrier* call). M_2 thus allows multiple nodes to actively write-share a page, thereby avoiding the penalties due to false sharing.

The implementation of M_2 is update based whereas the implementations of M_0 and M_1 are invalidation based. As a result, M_2 handles false sharing by allowing nodes to concurrently access different data items stored on a page. This is desirable for many applications. Unfortunately, the implementations of M_0 and M_1 do not lend themselves to such implementations. For instance, it is too expensive to track all writes in a software based implementation of M_0 . Similarly, invalidations are natural in the implementation of M_1 , which makes use of local consistency actions that only effect data cached at the node.

3.4 M_3 – User-directed Consistency

In M_3 , coherence activities make use of associations between shared data and the synchronization variables that control access to such data. In particular, only data associated with a synchronization variable S that is modified in a program region controlled by S is transmitted, and only to the processes that next acquire S . M_3 differs from M_2 because it sends changes to shared data to only the node that next acquires the synchronization variable. This determination is made at runtime based on the state of the queue associated with the synchronization variable. M_2 , on the other hand, sends the changes to all the nodes that cache the changed data. Thus, the approach taken in M_2 is similar to release consistency [11], whereas the one taken in M_3 is similar to entry consistency [7] and lazy release consistency [14]. However, lazy release consistency must propagate information about changes to all shared data whereas the explicit associations of M_3 allow it to send only the data that will be accessed in a synchronization region. In implementing M_3 , we had to extend the implementation of the synchronization constructs to include information about the data associated with a given synchronization variable. Compared to the previous three memory systems, it can be easily seen that M_3 incurs the absolute minimum amount of communication for state sharing.

4 Applications

We have implemented several applications on top of the memory systems M_0 through M_3 . We provide an overview of these applications in this section. The applications include three kernels from the NASA Ames NAS kernels, traveling salesperson (TSP), matrix multiplication (MM) and successive over-relaxation (SOR). A parallel application is characterized by the data access pattern, the synchronization pattern, the communication pattern, the computation granularity (which is the amount of work done between synchronization points), and the data granularity (which is the amount of data manipulated between synchronization points). The last two together define the task granularity of the parallel kernel. These attributes are as seen from the point of view of the individual processes executing the parallel application. If the application is implemented using the message-passing style, then the data access pattern becomes unimportant (except for any cache effects) since all data accesses are to private memory. Further, the synchronization pattern is usually merged with the communication pattern in such an implementation. On the other hand, if a shared memory style programming is used, the communication pattern is not explicit and gets merged with the data access pattern. In this section, we give an overview of these applications and discuss their performance on the various memory systems in the next section.

4.1 NAS Kernels

NASA Ames has developed a set of computationally intensive kernels derived from Numerical Aerodynamic Simulation (NAS) applications. We have implemented the following three kernels.

1. Embarrassingly Parallel (EP) kernel evaluates integrals by means of pseudo-random trials and is used in many Monte-Carlo simulations. As the name suggests, the kernel requires very little synchronization and communication among the parallel threads. Each process computes an equally large number of floating point random numbers and performs certain floating point operations on them. The only communication that happens is toward the very end when all the processes participate in a reduction operation to generate a global sum. The kernel has a very high task granularity and is expected to perform well irrespective of the type of the memory system.
2. Integer Sort (IS) kernel uses bucket sort to rank a set of 4M integers. The application executes in two distinct phases: in the first phase each process sorts a partition allocated to it and these sorted partitions are merged in the second phase. There is very little communication (non-local data access) in the first phase of the application, while the second phase involves considerable amount of data communication. The algorithm uses barrier synchronization between phases to synchronize the processes.

3. Conjugate Gradient Method (CGM) kernel computes the smallest eigen value of a sparse symmetric positive definite matrix (1400×1400). There are alternating phases of parallel and sequential parts in each iteration of this kernel. The computation intensive part of this kernel is the multiplication of this sparse matrix by a vector. The sparse matrix is represented using a row-start, column index format to reduce the amount of data communication during the vector-matrix multiplication. Each process is pre-assigned a set of rows of the sparse matrix to work on. Each process computes the elements of the result vector assigned to it with very little communication or synchronization with the other processes. This parallel part is followed by a sequential part that uses the result vector in a dot product operation. While there is considerable task granularity during the parallel part, the data movement for the serial part increases with the number of processes used in the algorithm.

In both IS and CGM kernels, there is considerable amount of true data sharing among the processes; and further in both kernels one data structure is accessed in sequence but another is accessed based on the value in the first data structure. Thus a considerable part of the data access patterns in both the kernels is data dependent.

4.2 Traveling Salesperson (TSP)

This application is unique because of the high degree of dynamic behavior of data sharing exhibited by it. Our implementation is for a 13 city tour and is similar to the one reported in [5]. It uses a branch-and-bound method. A set of partial tours are generated and processes evaluate these partial tours in parallel. They all share a work queue that stores the partial tours and the value of the best tour that has been found so far to discard bad tours. Thus, if the value of a certain tour being explored exceeds the current best value, the tour is abandoned and the process starts on another pending tour. The application completes when all tours have been explored.

4.3 Successive Over Relaxation (SOR)

SOR is an iterative method for solving discretized Laplace equations on a grid (size 128×128). The program is based on the parallel Red/Black SOR algorithm as described in [9]. The grid is partitioned among the processors and all the communication occurs between neighboring processors. Only the boundary elements of the grid need to be communicated between iterations.

4.4 Matrix Multiply (MM)

Lastly, we implemented everyone's favorite matrix multiply using input matrices of sizes 256x256. Each node computes a set of rows of the output matrix.

5 Performance Results

We have conducted detailed experiments to evaluate the performance of the six applications on the different memory systems. We first discuss the *completion times* of these applications when the number of nodes that execute the application are varied from 1 to 8. Completion time is the total execution time measured on a representative processor in the distributed system. The measurements are done with just the application executing on the system so there is no extraneous communication on the network. All the applications were run on SUN 3/60's connected over a 10 Mbits/s ethernet². The page-size, which is the unit of sharing in the memory systems $M_0 - M_2$, is 8K bytes.

To better understand the behavior of the applications and the memory systems, we also break down the completion time into component times to quantify the time spent in communication and synchronization operations. In particular, we define and measure the following component times:

- **Computation time:** The time spent by the process actually executing application code. Thus, during this time the process is not blocked waiting for synchronization, communication or consistency activities to complete. For an application that does not have time or data dependent behavior, this component must be the same on all the systems.
- **Synchronization time:** The time a process waits for synchronization operations to complete. These operations include *lock*, *unlock* and *barrier*. In M_3 data transfer is combined with synchronization upon a *lock* or a *barrier* call. Thus, data transfer time is also included in the synchronization time for M_3 .
- **Communication time:** In principle, this is the time spent in transferring data during the execution of the application. For the memory systems M_0 and M_1 , this is the elapsed time between the page fault trap and the subsequent return from the page fault handler after the page has been obtained and installed. In M_2 , communication time accounts for propagating modifications to data pages made in a synchronization region to nodes that are currently caching those pages. This includes computation of page *diffs*, sending the results to the manager and waiting for the manager to propagate the changes to all the nodes currently caching those pages.

²We discuss the impact of faster processors and networks in Section 6.

- **Network Handling time:** This time is spent in responding to network messages (for example, invalidation and forward requests for the data pages). If the representative processor is currently waiting for a communication or synchronization operation at the time of receiving such a network message, then the ensuing network handling times gets absorbed in the communication or synchronization time respectively.

We first discuss the completion times. The component times are discussed in the following subsection.

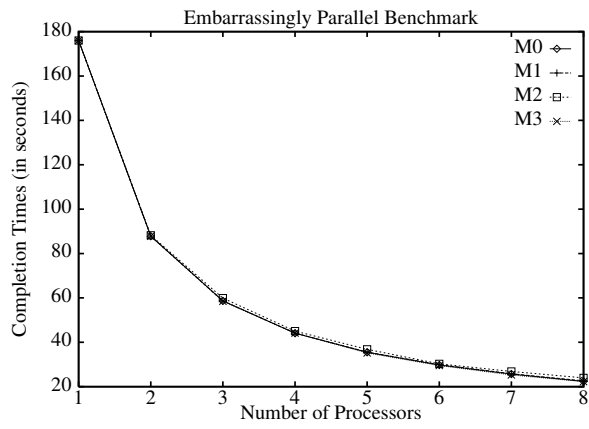
5.1 Completion Times

Figure 1 shows the completion times for each of the applications discussed in the previous section. These times are plotted against the number of nodes. The times for one node are for sequential implementations of the applications and do not include any synchronization overheads. Since M_3 takes advantage of application specific information (association of data with a *lock*, etc.), it performs well and close to the ideal execution time³ which assumes communication costs to be zero. This supports our assertion that M_3 captures the true cost of state sharing. EP and matrix multiply applications perform well on all the DSM systems. The performance of the various DSM systems is also similar for IS because the large sequential component makes it impossible to achieve significant speedups for any of the protocols (about 1.7 for M_3 with 8 nodes). In general, the underlying DSM system does not have a significant impact on the overall performance of an application if one or more of the following attributes hold for an application:

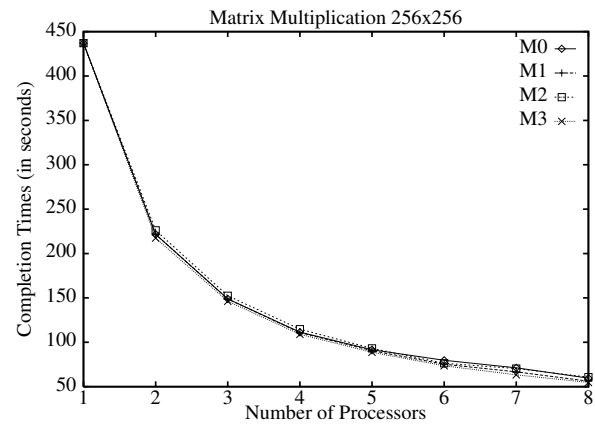
1. The data access pattern of an application is such that there is either very little sharing or mostly read-sharing and hence does not result in much communication.
2. The computation granularity of an application is sufficiently large.
3. The writes to shared data display a high spatial locality and thus do not interfere with the activities on other nodes.

All of the above three attributes are true for MM, and attributes 1 and 2 are true for EP. For the other applications, these attributes may hold for some part of the application. However, either the sequential parts in the application, or the inherent inability of a given DSM system to exploit the nature of the application results in poor performance. We discuss three of the applications CGM, TSP and SOR in detail because their completion times show differences for the four DSM systems.

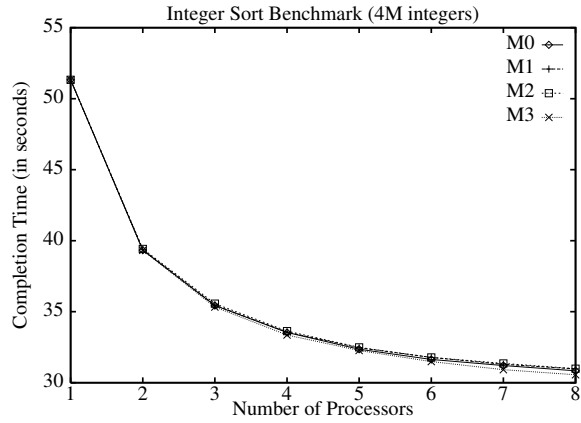
³The ideal execution time is $t_s + t_p/n$ where t_s and t_p are the times spent in the sequential and parallel components on one node, and n is the number of nodes that execute the application.



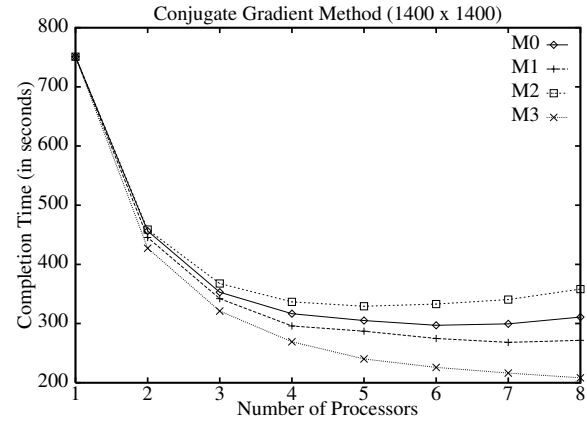
(a) Embarassingly Parallel



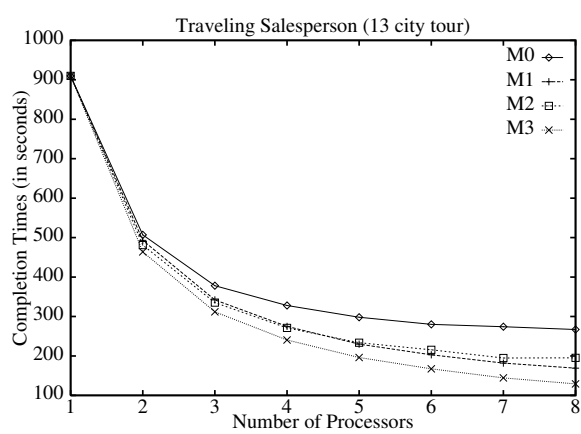
(b) Matrix Multiplication



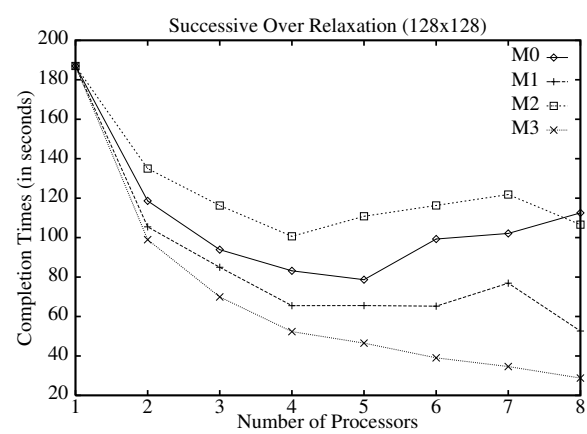
(c) Integer Sort



(d) Conjugate Gradient Method



(e) Traveling Salesperson



(f) Successive Over-Relaxation

Figure 1: Completion Times

In conjugate gradient (CGM), 90% of the time (in a sequential implementation of the algorithm) is spent in a sparse matrix multiplication which is efficiently parallelizable. Yet, as can be seen from Figure 1(d), the performance for even M_3 tends to saturate beyond about 5 nodes. The sequential part that follows the parallel phase requires bringing all the data to one node which is the main source of this saturation. The result vector generated in the parallel phase fits entirely in one page for the problem size considered (1400 elements). M_0 does not perform well due to the false sharing that is inherent in invalidation based schemes. This leads to the page moving between the nodes where it is written. Although M_1 is also vulnerable to false sharing, the invalidations are local to each node as opposed to M_0 which invalidates all cached copies, and hence its performance is better than M_0 . M_2 , while permitting concurrent writers during the parallel phase, incurs the overhead of propagating modifications made at a node to all other nodes which limits its performance for this application. Since all nodes where a page is modified do this propagation via the same manager node, there is contention at the manager which makes the propagation times longer. This explains why M_0 , which serializes the writers due to false sharing, still performs better than M_2 . M_1 performs better than M_0 because of local invalidations which results in 7% fewer page faults (and corresponding messages). Also, it allows readers to co-exist with a writer. M_3 has about 25% lower completion time than M_1 when eight nodes execute the application.

In the case of TSP, false sharing or unnecessary updates are not a problem. However, the page oriented nature of M_0 results in excessive communication overhead for propagating the changes to the new tour value (a single integer). This explains the poor performance of M_0 compared to other protocols. M_2 performs better than M_0 in this application because of two reasons. First, it propagates a new tour value found by a process to other processes without waiting for them to execute a synchronization operation which reduces the exploration of bad tours. Also, in the *diff* operation, we do not send the XOR result of the entire page when no changes are made to the page data after a certain offset in the page. This makes the message sizes small for TSP. Despite the page oriented nature of the protocol that implements M_1 , the inherently asynchronous nature of the algorithm allows it to use old tour value thus reducing the number of messages compared to M_0 and M_2 . M_3 provides the best performance because it has lower processing overheads and sends much less data (M_3 sends 18K bytes whereas M_1 sends over 20M bytes).

In SOR, M_2 does not perform well due to the same reasons that we had in CGM (propagating modifications to other nodes on exiting from a synchronization region). However, its relative performance is better for SOR than CGM because a node communicates only with its neighbors and hence the modifications only need to be propagated to these nodes. M_1 does better than M_0 because it allows readers to co-exist with a writer. This benefits it because of the inherent write/read false sharing in SOR. The execution times increase from 4

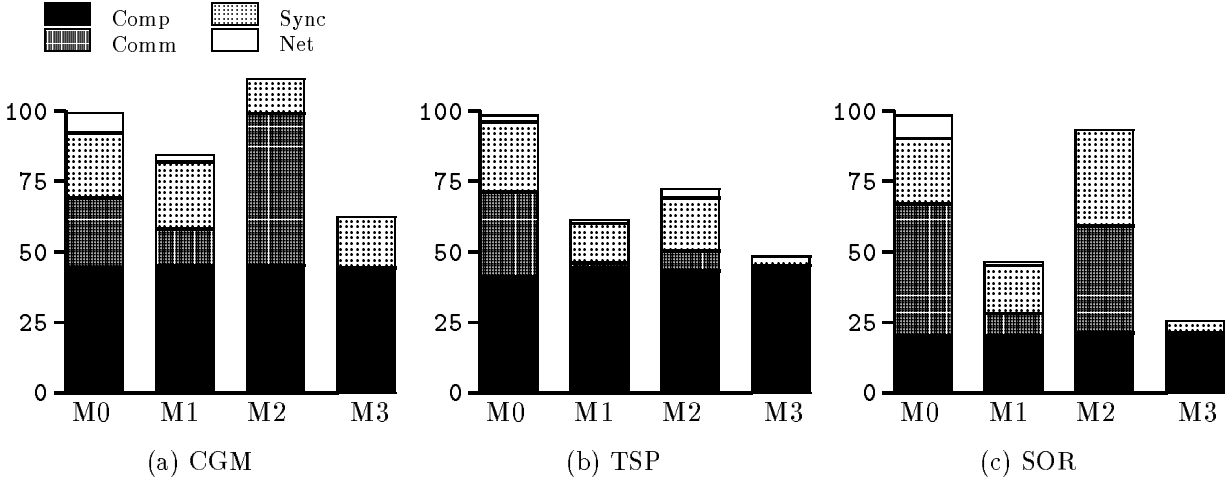


Figure 2: Component Times (scaled to 100 for M_0)

to 7 nodes for the page based protocols because with these many nodes, the data partitioning results in some nodes having to incur more communication overhead than others to access their respective data partitions.

5.2 Breakdown of Completion Times

In order to understand the differences among the DSM implementations, we also measured the component times for the three applications, CGM, TSP and SOR, which have different completion times on the different DSM systems. We show the execution profile of a representative node in Figure 2 when the applications are executed on 8 nodes. We have scaled the completion time to 100 for the memory system M_0 .

The execution profiles shown are for node 0. For TSP and SOR, all the nodes executing the application have similar execution profiles. In CGM, node 0 executes the sequential part of the kernel as well as the parallel part. While node 0 executes the sequential part, the other nodes are blocked at a barrier. So the execution profiles of the other nodes have a smaller computation component and a higher synchronization component due to the waiting time at the barrier.

- **Computation Time** - In CGM and SOR, the execution of the application is not data dependent. As a result, all four DSM systems have the same computation time for these applications. TSP execution is data dependent since the best tour value is used to prune bad tours. In M_0 , a new value of best tour propagates to other nodes as soon as the new value is written. This happens as the cached values at other nodes are invalidated and these nodes fault and fetch the new value when they next access this variable. In M_1 , a node could work with an old value of best tour because a write does

not result in communication with other nodes. Only when a node starts computing a new tour value, the old value of best tour will get invalidated and a future access will result in the new value to be brought to the node. As a result, the computation time is more for M_1 in the case of TSP.

- **Communication Time** - Of the component times, the communication time is the one that is of most interest in this study because state sharing methods attempt to minimize this component. In the case of M_3 , the modified data is transferred with the associated synchronization variable. Thus communication and synchronization are inter-twined and lumped into one component, namely, the synchronization one in M_3 . This is seen in the results presented in Figure 2. One interesting observation is that M_1 , the system based on weak consistency, has significantly lower communication time compared to M_0 and M_2 . This is because M_1 uses local invalidations and hence consistency actions do not result in communication. In general one would expect higher communication time for M_0 , owing to the increased number of page faults incurred due to invalidations on write operations. In M_2 , modified data is sent by each node only prior to exiting a synchronization region. In spite of this, M_2 displays the highest communication time of all the memory systems for CGM due to two reasons: first there is a large number of messages ($O(n^2)$) as each node has to send its modified data to all the other nodes, and second there is increased contention at the data manager (which has to propagate the diffs) due to the simultaneity of these messages. For TSP, communication time is the highest for M_0 as expected. M_1 has very low communication time because it works with old tour values and hence updating the tour value does not result in communication. The communication time for M_2 is higher than M_1 but significantly lower than M_0 even though the best tour value has to be sent to all the other nodes. This is because the size of shared data is very small (e.g., 4 bytes for best tour) and M_2 sends only the *diffs* thus resulting in very small messages. Unlike CGM, SOR requires only near neighbor communication. Thus we would have expected that M_2 would incur significantly less communication compared to M_0 for SOR. However, we observe that the communication times for both systems are very close for SOR. This is because every alternate location in a page gets modified in each iteration thus negating any advantage of using diffs for reducing the size of the messages. M_1 has much lower communication time since it allows a single writer to be concurrent with multiple readers thus handling the write-read false sharing which is inherent in the SOR application.
- **Synchronization Time** - The synchronization time component accounts for the time a process spends while waiting for a synchronization operation to complete. Thus, this time not only depends on the cost of communicating with a remote synchronization server but also on contention for a synchronization variable (e.g., a *lock*). Although all

systems make the same number of synchronization calls (except in TSP where execution is data dependent), contention for the lock, which depends on server load and the time a process is inside the critical section (which could be quite different for the protocols due to the process page faulting inside the critical section), makes synchronization time quite different for the various systems. M_3 , has the lowest synchronization time due to lower contention since processes never page fault while holding locks. M_1 has the extra overhead of doing the local invalidations on acquiring a synchronization variable. The large synchronization component for M_2 in SOR is because of contention at the manager node which propagates the diffs. Thus a process arriving at the barrier has to wait until all other nodes have propagated their changes.

- **Network Handling Time** - The network handling time, spent in processing request messages while a node is executing application code, is the highest for M_0 . This is to be expected because nodes may get requests to forward data pages and invalidation messages while the application code is executing at the node. In M_2 , the update messages are mostly received when the receiver is also waiting for its modifications to be propagated (as in the case of CGM and SOR where all the processes arrive at the barriers around the same time). Thus, the time spent in processing the message is not accounted in network handling time. M_1 has small network handling time for all applications since it does not require sending invalidation messages.

In summary, we see that weaker consistency (M_1) does provide some performance improvements over M_0 because it has much lower communication costs. M_2 , because of the excessive communication costs due to propagation of modified pages to all nodes that cache these pages, does not perform as well as expected. There are other reasons why M_0 - M_2 , the memory systems that require no or minimal assistance from the user, do not match the performance of M_3 in which consistency activities are user-directed. One reason is the mismatch in the unit of consistency maintenance; it is a page (8K bytes) for the first three protocols. In spite of this, since only *diffs* have to be propagated instead of entire pages, M_2 has an edge in certain applications such as TSP which offer the scope for reducing the message sizes. In M_3 , only the data that will be accessed in the synchronization region controlled by the synchronization variable is sent. Thus, it further reduces the size of the data message. Also, due to the explicit association of data with synchronization variables in M_3 , no processing overhead is incurred (e.g., copying and XOR operations in M_2) in identifying the data items that need to be transferred to a node when it acquires a synchronization variable.

6 Discussion

We discussed the execution behavior of the chosen applications with different memory systems in the previous section. There are several other observations we would like to make about this study in general.

Appropriateness of Applications: The NAS kernels come from the domain of numerical applications and several applications having similar characteristics (e.g. set of linear equations, partial differential equations etc.) have been studied in DSM systems [6, 18]. In addition, the TSP and SOR applications have been investigated extensively in evaluating DSM systems. We believe these applications do capture the data sharing patterns of many applications that can be run on distributed platforms.

Programmability of Applications: The complexity of programming the applications on the four systems was not appreciably different. The weaker consistency of M_1 did not require changes in the programs because the synchronization already required for M_0 establishes enough causal dependencies to ensure consistency of data. M_2 only required the identification of synchronization operations to send updates immediately prior to release operations which was straightforward. Although the user-directed scheme required the most programming effort because of the explicit association between synchronization variables and data items, determination of these associations was not complex for the applications.

Data and Computation Granularity: The large overhead associated with communication between nodes of a distributed system implies that good performance is only possible when there is sufficient computation granularity (i.e., the work done by each node between communication events). This indeed was seen in our experiments. For example, when we ran CGM (size 14000 x 14000) and SOR (size 512 x 512) [12], better speedups were observed for all the memory systems. However, there are differences between the memory systems when it comes to data granularity (the amount of shared data processed between communication events). The mismatch between the large page size (8K) and smaller data granularity for some of the applications explains the gap in the execution times of these applications between the first three systems (which are page based) and M_3 . We believe that a smaller page size would have been better suited for M_0 , M_1 and M_2 for the applications we studied.

User Directives or Annotations: We have argued that explicit user directives are the reason for the superior performance of M_3 . However, some optimizations were done for each system to obtain the best performance with that system. For example, updates were not sent to nodes that did not need them in M_2 and *readonly* data was tagged to reduce the size of version vectors for M_1 . These optimizations are fundamentally different from the directives of M_3 or the annotations used in Munin [6]. This is because correct execution of the application does not depend on these optimizations. We believe such optimizations can be easily automated in the compiler and do not require any change in the application programming. On the other

hand, in M_3 , incomplete associations will lead to incorrect executions. Thus, when we claim that user directives can only provide the best performance, these directives will require a significant understanding of the data sharing patterns of the applications.

Impact of Faster CPU and Networking: Our experimental test-bed consisted of Sun 3/60 machines connected by a 10 Mbits/s ethernet. An obvious question is if the differences in the memory systems would persist with faster processors or when the nodes are connected by a high speed network such as an ATM. It can be easily argued that the results will still be valid with increased CPU speed because all the curves will shift down. However, faster network will make the difference among the memory systems less significant. This is because the protocols that send large messages (pages) will benefit more from the increased network speed [21].

7 Concluding Remarks

A wider acceptance of the DSM abstraction as a state sharing mechanism in distributed systems and the numerous approaches for implementing it motivated our work. We implemented the protocols in the operating system using a uniform set of mechanisms to make the comparisons more accurate and meaningful. By quantifying the overhead associated with the protocols we have been able to distill several principles. First, although weaker consistency and exploiting synchronization information can provide performance improvements over M_0 , user directives or annotations are needed to eliminate unnecessary data transfer and to reduce processing overheads. Such associations can be done for user defined data items which can reduce the problems of false sharing and of having to move data in units of pages.

References

- [1] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. Technical Report CS-1051, University of Wisconsin, Madison, September 1991.
- [2] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *11th International Conference on Distributed Computing*, May 1991.
- [3] R. Ananthanarayanan, M. Ahamad, and R.J. LeBlanc. Coherence, synchronization and state-sharing in distributed shared memory applications. In *Proceedings of the 1993 International Conference on Parallel Processing*, 1993.
- [4] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report Report RNR-91-002, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, January 1991.
- [5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Experience with distributed programming in Orca. In *In Intl. Conf. on Computer Languages*, 1990.
- [6] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–177, March 1990.
- [7] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.
- [8] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, (9, 3):272–314, 1991.
- [9] J. S. Chase, F. G. Amador, H. M. Levy E. D. Lazowska, and R. J. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th Symposium on Operating System Principles*, 1989.
- [10] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the ACM Symposium on Operating System Principles*, 1989.
- [11] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

- [12] Ranjit John and Mustaque Ahamad. Evaluation of causal distributed shared memory. Technical Report GT-CC-94-34, Georgia Institute of Technology, 1994.
- [13] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb 1988.
- [14] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Computer Architecture News*, 20(2), May 1992.
- [15] R. E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.
- [16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [17] J. Lee and U. Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, 1991.
- [18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [19] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.
- [20] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Eighteenth Annual International Conference on Parallel Processing*, August 1989.
- [21] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2), May 1993.