Implementation and Performance of DSMPI

LUIS M. SILVA,[1] JOÃO GABRIEL SILVA,[1] AND SIMON CHAPPLE[2]

[1]*Departamento Engenharia Informática, Universidade de Coimbra-POLO II, Vila Franca-3030 Coimbra, Portugal; e-mail: luis@dei.uc.pt*

[2]*Quadstone Ltd., 16 Chester Street, Edinburgh, EH3 7RA, Scotland; e-mail: src@quadstone.co.uk*

ABSTRACT

Distributed shared memory has been recognized as an alternative programming model to exploit the parallelism in distributed memory systems because it provides a higher level of abstraction than simple message passing. DSM combines the simple programming model of shared memory with the scalability of distributed memory machines. This article presents DSMPI, a parallel library that runs atop of MPI and provides a DSM abstraction. It provides an easy-to-use programming interface, is fully, portable, and supports heterogeneity. For the sake of flexibility, it supports different coherence protocols and models of consistency. We present some performance results taken in a network of workstations and in a Cray T3D which show that DSMPI can be competitive with MPI for some applications. © 1997 John Wiley & Sons, Inc.

1 INTRODUCTION

Distributed shared memory (DSM) systems provide the shared memory programming model on top of distributed memory systems (i.e., distributed memory multiprocessors or networks of workstations). DSM is appealing because it combines the performance and scalability of distributed memory systems with the ease of programming of shared memory machines. In the DSM paradigm, processes communicate with each other through shared variables that are placed somewhere in the system, but the programmer does not have to worry about where the data are. With message passing, the programmer has to be aware of the data movements between processes, and each process has to know when to communicate, with whom to communicate, and what data should be sent in a message. There are many algorithms for which the message-passing implementation is nontrivial, tedious, and error prone. The DSM system hides the remote communication from the application programmer and provides a simpler abstraction that the programmer understands well. As a consequence, the code written for DSM becomes more compact and easy to read than the same code written for message passing. Moreover, DSM facilitates the porting of existing sequential codes based on concurrent programming to distributed memory machines.

DSM has received much attention in the past decade and several DSM systems have been presented in the literature [1–3]. Basically, they can be classified in four main approaches:

1. Hardware implementations, where the cache coherency and DSM protocols are supported by the hardware. Examples include the DASH multiprocessor [4], PLUS [5], and the KSR-1 machine [6].
2. Operating system implementations, which extend the virtual memory-management mechanisms to access remote data. Some relevant examples are the IVY system [7], Mirage [8], and Clouds [9].

Received September 1995
Revised March 1996

© 1997 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 6, pp. 201–214 (1997)
CCC 1058-9244/97/020201-14

3. Compiler implementations, where all the data distribution and management is implemented by the compiler and a run-time system. Interesting examples of languages are Orca [10], Amber [11], and COOL [12].
4. Library implementations. In this approach, the DSM abstraction is completely implemented by a run-time library that is linked with the application program. Systems like Munin [13], TredMarks [14], and ADSMITH [15] can be included in this category.

DSMPI is also included in this latter class. It is a parallel library implemented on top of MPI whose main aim is to provide an easy-to-use programming interface based on the abstraction of a globally accessed shared memory that can be read and/or written by any process of an MPI application. The most important guidelines that we took into account during the design of the library were to

1. Assure full portability of DSMPI program.
2. Provide an easy-to-use and flexible programming interface.
3. Support heterogeneous computing platforms.
4. Optimize the DSM implementation to allow execution efficiency.

The library was implemented without using any special features of the underlying operating system. It does not require any special compiler, preprocessor, or linker. This is a key feature to obtain portability. In this point, we depart from the approach that was followed by other library-level implementations of DSM, like Munin [13] or TreadMarks [14], because those systems make use of some memory-management facilities provided by the underlying operating system. Munin, for instance, requires a special preprocessor and a modified linker and uses the memory-management support provided by the experimental system where it was implemented (V Kernel). TreadMarks offers more portability because it is a user-level library that runs on top of UNIX and uses a standard UNIX compiler and linker. However, it still relies on the memory page protection mechanism of UNIX to detect accesses to the shared pages. Those features provided by UNIX are not available in other native operating systems of multiprocessor machines. Those two systems have the nice advantage of providing transparent access to the shared data, but at the expense of some lack of portability.

In our case, we sacrificed full transparency to achieve portability. This means that all shared data and the read/write operations should be declared explicitly by the application programmer. The sharing unit is a program variable or a data structure. For this reason, DSMPI can be classified as a structure-based DSM system (like ADSMITH [15]) as opposed to other DSM systems that are implemented in a page basis and make use of the operating system virtual memory facilities. Page-based DSM systems (like IVY [7] and Mirage [8]) are prone to the problem of false sharing. When different processes want to write different variables that are located in the same page, the system has to send the page back and forth between them resulting in unnecessary coherence traffic. Munin and TreadMarks alleviate the problem of false sharing by allowing multiple writers to write to the same page. This was achieved through a complex scheme to merge the changes at the synchronization points.

DSMPI does not provide full transparency because it does not make use of any memory-management facility of the operating system, nor does it require the use of any special compiler. It does not incur in the problem of false sharing because the unit of shared data is completely related to existing objects (or data structures) of the application. At the same time, it allows the use of heterogeneous computing platforms. Recently, it has been concluded by several researchers that it is highly desirable to integrate heterogeneous hosts into a coherent computing environment to support parallel applications. Heterogeneity is quite easy to support in structure-based (e.g., DSMPI and ADSMITH [15]) or object-based DSM systems (e.g., Agora [16]), but rather more complicated in page-based DSM systems. Mermaid [17] is an example of a system that supports heterogeneous DSM: Memory is shared in pages, but each page can only contain one type of data. This is a enormous drawback and seems to be the only way to support heterogeneity in page-based DSM systems.

DSMPI is a heterogeneous DSM system, whereas most of the other systems are limited to homogeneous platforms. This is in fact one of the most interesting features of our system. Because the library knows the exact format of each shared data object (data type and number of elements), it becomes quite straightforward to support heterogeneity by making use of the MPI features to support heterogeneous computing.

ADSMITH [15] is another DSM system that provides support for heterogeneity. It was implemented on top of PVM and has some similarities with our system. The main differences are that DSMPI provides more protocols of replication and different models of consistency and support for fault-tolerance. Finally, DSMPI allows the coexistence of both programming models (message passing and shared data) within the same application. This has been considered recently

as a promising solution for parallel programming [18–20].

Concerning absolute performance, we can expect applications that use DSM to perform worse than their message-passing counterparts. However, this is not always true. It really depends on the memory-access pattern of the application and on the way the DSM system manages the consistency of replicated data.

We tried to optimize the accesses to shared data by introducing three different protocols of data replication and three different models of consistency that can be adapted to each particular application in order to exploit its semantics. With such facilities we expect DSM programs to be competitive with MPI programs in terms of performance. Some performance results collected so far corroborate this expectation.

The rest of this article is organized as follows: Section 2 describes the general organization of DSMPI. Section 3 presents the replication protocols provided by the library, whereas Section 4 describes the models of consistency that were implemented. The programming interface is presented in Section 5. Section 6 shows some performance results and Section 7 concludes the article.

2 DESIGN OVERVIEW

In DSMPI there are two kinds of processes: application processes and daemon processes. The latter are responsible for the management of replicated data and the protocols of consistency. Because the current implementations of MPI [21] are not thread safe, we had to implement the DSMPI daemons as separate processes. This is a limitation of the current version of DSMPI that will be relaxed as soon as there is some thread-safe implementation of MPI. All the communication between daemons and application processes is done by message passing. Each application process has access to a local cache that is located in its own address space and where it keeps the copies of replicated data objects. The daemon processes maintain the master copies of the shared objects. DSMPI maintains a two-level memory hierarchy: a local cache and a remote shared memory that is located in and managed by the daemons.

The number of daemons used by an application is chosen by the programmer as well as their location, by using a configuration file (`dsmconf`) that is read by the DSMPI initialization routine. In this way, the user has the freedom to choose the most convenient mapping of daemon processes according to the application needs. In practice, there is a notion of virtual domains where each daemon is responsible for one or more application processes. One of the daemons is elected as the master daemon and will be responsible for some operations that require some centralized responsibility.

The ownership of the data objects is implemented through a static distributed scheme. While a centralized scheme would introduce a bottleneck, a dynamic distributed policy would require the use of broadcasts or forward messages to determine the current owner of a data object [22]. In the static distributed policy, the owner daemon of each object is chosen by the runtime system during the startup of the application and remains fixed during the lifetime of the application. Each process maintains a local directory containing the location of each object in the system. This static distribution strategy requires less control messages than the dynamic strategy and does not introduce the congestion of a central server.

The mapping of data objects to daemons must be chosen by the application programmer. When the process creates a shared object, the assigned owner will be the associated daemon. When correctly used, this facility can enhance the performance of the applications. However, if the mapping provided by the programmer is completely inadequate, it would certainly degrade the performance of the application. The next version of DSMPI will have an adaptive mapping strategy based on run-time heuristics that will adapt the mapping of the data objects to daemon processes according to the observed communication patterns of the application. Such facility could alleviate or adjust some wrong decision taken by the application programmer. The current version of DSMPI is still limited in this aspect.

3 DATA REPLICATION PROTOCOLS

The library allows the programmer to chose the replication strategy for each shared object among three possible choices:

1. WRITE_MOSTLY
2. READ_WRITE
3. READ_MOSTLY

The first class uses a single copy of the object, whereas the two other classes replicate the object. The replication of data is only effectively justified if the number of reads is higher than the number of writes. Data replication is one way of exploiting parallelism, because multiple reads can be executed in parallel.

The first class (`WRITE_MOSTLY`) represents those objects that are frequently written. For these kinds of

objects it is not worthwhile to replicate them among the caches of the processes. Only one copy is maintained by one of the daemons.

The second class (READ_WRITE) includes those objects that have roughly the same number of read and write requests. These objects are replicated among the caches of the processes that perform some read request on them. There is one daemon that keeps the primary copy of the object and maintains a copy-set list of all the processes that have a copy in its local cache. A process is included in that list when it issues a remote read request. After that and in the absence of write operations, the process reads the object from its local cache. That daemon is responsible for the consistency of the replicated object, and for this class of object the library uses a write-invalidation protocol. This means that on a write operation all the cached copies of the object are invalidated from the local caches. Only the process that writes to the object maintains an updated copy in its local cache. When another process wants to read or write that object, it gets a cache-miss, and has to fetch the object remotely from the daemon that maintains the primary copy.

The objects that belong to the third class (READ_MOSTLY) are also replicated among the caches of the processes that use them. These objects have a higher ratio of read over write requests, and in this case the library uses a write-update protocol. All the cached copies of the object are updated atomically after each write operation. This scheme does not incur in cache-misses as the write-invalidation protocol but in the case of large size objects it requires the sending of large size messages that carry the update notices.

Both protocols have advantages and disadvantages [22, 23]. Invalidation-based protocols have the advantage of using smaller consistency-related messages than update-based protocols since they only specify the object that needs to be invalidated and not the data itself. The write-update protocols reduce the likelihood of a cache-miss. The choice between these protocols depends on some factors [23] like

1. The ratio of read/write accesses to the shared object
2. The size of the data object
3. The overhead of the update messages

The application programmer has the freedom to choose among those three classes when it creates a particular shared object. It has been argued in [22] that no single algorithm for DSM will be suitable for most applications. Shared objects have different characteristics, thus it is important to select the coherence protocols that best match the application characteristics.

The replication of data is an important feature to improve the efficiency of the application. DSMPI is not the only system that provides more than one class of objects. Munin [13] is another system that provides several classes of objects. Through the use of annotations, the programmer can choose among seven different coherence protocols tailored for different types of data. The type of the data objects remains static once specified. Munin cannot dynamically adjust the coherence protocols to possible changes in the behavior of the application. The same happens with the current release of DSMPI.

The study presented in [24] considers multiple coherence protocols. It presents a self-adjusting scheme where the run-time system monitors the usage pattern of each data object and performs a protocol adjustment to the objects that have a different behavior than was specified previously through the user annotations. It has been proved to be a very interesting technique with quite good results.

Another example is the ORCA language [25] that has an extended compiler which determines the access patterns of the processes to the shared objects and forwards that information to the run-time system, that decides which objects to replicate and where to store single copy objects. This has been proved to result in a better overall performance.

We plan to incorporate in the next version of DSMPI a similar scheme based only on run-time heuristics. Besides, the frontier among those three class of objects is not always quite clear and the hints provided by the programmer can be far from the optimum decision. Such cases should be corrected by the run-time system during the execution of the program.

4 MODELS OF CONSISTENCY

In order to further improve efficiency, we have implemented three different models of consistency that apply only to READ_WRITE and READ_MOSTLY objects:

1. Sequential consistency (SC)
2. Release consistency (RC)
3. Lazy release consistency (LRC)

The SC model was proposed in the IVY system [7], where all the operations in shared data are performed as if they were following a sequential order. This is the normal model of consistency, which in some particular

cases does not perform very well in DSM systems implemented on networks like Ethernet.

For such cases, the programmer can make use of the RC model, which in our case implements the protocol of the DASH multiprocessor [4]. In this model, it is assumed that all the write accesses to shared objects are protected by synchronization primitives (acquire and release, corresponding to lock/unlock operations). It attempts to mask the latency of write operations by allowing them to be performed in the background with the computation. The process is only stalled when executing a release, at which time it must wait for all its previous writes to perform. However, this model does not attempt to reduce the number of messages exchanged. Rather, it only tries to reduce the latency of write operations.

Another RC model has been implemented in Munin [14], where all the write notices from a process are buffered and only sent when the processor reaches a synchronization release point. This does not introduce a significant improvement over the DASH release consistency protocol, and by this reason we chose the previous one.

To reduce the message traffic, the system can use the LRC model of consistency. DSMPI implements a similar protocol as proposed in the TreadMarks system [14]. This model relaxes the RC protocol by further postponing the sending of write notices until the next acquire. When a process writes into a replicated data object, rather than sending update/invalidate notices to all the processes that belong to the copy-set list, it keeps a record of the objects that it updated. Then the owner of the lock notifies the acquirer of which objects have been modified, causing the acquirer to invalidate its local copies of these objects.

The LRC schemes keeps track of the causal dependencies between write operations, acquires and releases, allowing it to propagate the write notices lazily only when they are needed. To implement this scheme we have used the Vector Time mechanism, and the resulting protocol, albeit complex, reduces considerably the number of messages and the amount of data exchanged on behalf of the replication protocols [26, 27].

Another relaxed memory model — entry consistency (EC) — was proposed in the Midway system [28]. This model requires all shared data to be explicitly associated with some synchronization variable. As a result, when a process acquires a lock or semaphore, the EC-based DSM system only needs to propagate the updates to the shared data associated with that lock or semaphore. It requires more effort from the application programmer, but simplifies the implementation of the consistency protocol.

```
- DSMPI_Startup(MPI_Comm *comm);
- DSMPI_Exit(void);
- DSMPI_Decl_SO(char* name, int count, MPI_Datatype type, DSM_SO *shobj);
- DSMPI_Create_SO(DSM_SO shobj, int access_type);
- DSMPI_Read(DSM_SO shobj, void *buf, int model);
- DSMPI_Write(DSM_SO shobj, void *buf, int model);
- DSMPI_ReadSome(DSM_SO shobj, void *buf, int count, int offset, int model);
- DSMPI_WriteSome(DSM_SO shobj, void *buf, int count, int offset, int model);
- DSMPI_Init_Lock(int lock_id);
- DSMPI_Lock(int lock_id);
- DSMPI_UnLock(int lock_id);
- DSMPI_Init_Sem(int sem_id, int value);
- DSMPI_Wait(int sem_id);
- DSMPI_Signal(int sem_id, int N);
- DSMPI_Barrier(int barrier_id, int N);
- DSMPI_Restart(void);
- DSMPI_Pack_chkp(void *ptr, int count, MPI_Datatype dt_type);
- DSMPI_Checkpoint(void);
```

FIGURE 1 DSMPI primitives.

5 PROGRAMMING INTERFACE

The library provides a C interface and the programmer calls the DSMPI functions in the same way it calls any MPI routine. The complete interface is shown in Figure 1.

5.1 Initialization and Termination

To start a DSMPI application, all the processes must invoke `DSMPI_Startup()` collectively. This function returns an MPI communicator that can be used by the application processes to communicate directly through message passing. To terminate correctly a DSMPI application, all the processes have to call `DSMPI_Exit()`.

5.2 Declaration and Creation of Shared Objects

The set of shared objects used by the application should be declared explicitly by all the processes at the beginning of the application through the routine `DSMPI_Decl_SO()`. With this routine, the programmer has to specify the data type of the object and the number of elements. It returns a shared object identifier that should be used in any further reference to that object. To create one shared object, a process has to call the `DSMPI_Create()` primitive. In this call, the programmer has the freedom to choose the replication strategy for the shared objects by indicating the access type among the three options: `WRITE_MOSTLY`, `READ_WRITE`, and `READ_MOSTLY`.

5.3 Read and Write Operations

Communication between processes is made by using explicit read and write operations on shared objects:

The programmer makes use of DSMPI_Read() or DSMPI_Write() and does not need to be aware of the object location. An object can be written/read locally from its own cache, or remotely from its primary copy. In the current version, the programmer has the freedom to specify the required model for data consistency among three choices: DSM_SC, DSM_DRC, and DSM_LRC, where DSM_SC corresponds to sequential consistency, DSM_DRC to the dash release consistency protocol, and finally DSM_LRC corresponds to lazy release consistency.

To read or write only parts of some object, the programmer may find the following routines very useful: DSMPI_ReadSome() and DSMPI_WriteSome(). The programmer has only to specify the offset within the object and the number of basic elements he wants to read (or write). These routines have been proved to be very effective to operate in large size objects (like arrays and vectors) when the program only wants to read or update a small part of the object.

5.4 Synchronization Primitives

To control the concurrent access to shared variables, the programmer has to use some synchronization primitives. DSMPI provides locks DSMPI_Lock(), DSMPI_UnLock(), semaphores DSMPI_Wait(), DSMPI_Signal(), and barriers DSMPI_Barrier(). These are the usual synchronization primitives provided by most DSM systems.

5.5 Support for Checkpointing

The library also provides support for checkpointing through the use of three additional primitives: DSMPI_Restart(), DSMPI_Pack_chkp(), and DSMPI_Checkpoint(). The first routine should be placed inside the program after the initialization part. DSMPI_Pack_Chkp() is used by the programmer to specify which private data should be saved in each checkpoint operation. The programmer is also allowed to determine which shared data should be included or not in the checkpoint contents. Finally, the DSMPI_Checkpoint() routine should be called periodically and be placed in some points that correspond to a global consistent state of the application. More details about the checkpointing scheme and some performance results can be obtained in [29].

6 PERFORMANCE RESULTS

The main question to be answered is: What is the price in terms of performance the users of DSMPI have to pay when compared with MPI? Applications that use any DSM system are expected to perform worse than if they use message passing directly because DSM is implemented on top of message passing. However, there are some studies [30] which show that DSM can be competitive with message passing for many applications. We also present some results that corroborate that study.

Ideally, the number of messages exchanged in a software implementation of DSM should be equal to the number of messages exchanged in a message-passing implementation of the same application. This hardly happens, but the DSM system should be optimized to achieve a competitive level of performance. The performance of a DSM system is directly affected by several factors, such as:

1. The memory-access pattern of the application
2. The replication of shared data and the protocol that assures its consistency
3. The implementation of relaxed models of consistency
4. Possible optimizations that try to reduce the exchange of DSM data through the communication network

The performance results are presented in two different sections: Section 6.1 presents a detailed study of the data replication protocols and the models of consistency, whereas Section 6.2 presents a comparison between the performance of DSMPI and MPI.

6.1 Performance of Data Replication Protocols and Models of Consistency

In this section, we present some performance results that represent the behavior of data replication for different models of consistency and application characteristics. The benchmark we used was a quite simple synthetic benchmark: It is represented by six processes that are continuously reading and writing into a shared object that is protected by a lock. It represents the worst-case benchmark because the processes do not perform any computation at all. They are just in an iterative cycle reading and writing the shared object.

Each slave process executes for N iterations. We made several experiments by changing the ratio of read operations over write operations (R equal to 1/1 and 3/1) and the size of the shared object (S equal to 10, 1K, 100K integers). The performance results were taken on a network of Sun4 workstations connected by a 10 Mb/s Ethernet. All the execution times are given in seconds.

To directly compare the invalidation with the up-

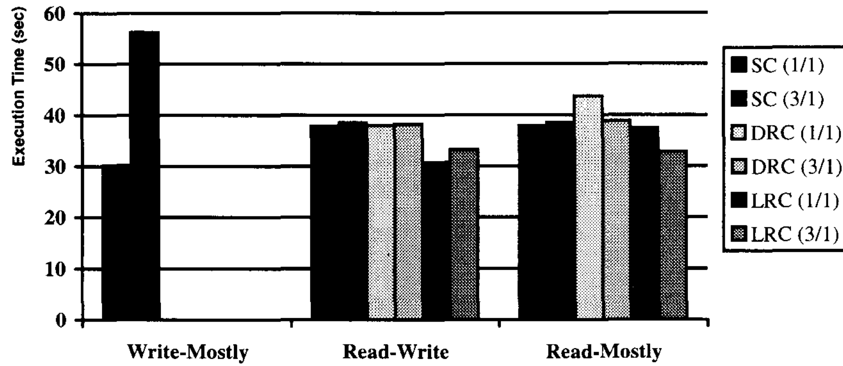


FIGURE 2 Performance results for small-size objects (S = 10).

date protocol. we condensed some of the results in the next four figures. For instance, Figure 2 presents the execution time to the benchmark when using a small-size shared object (i.e., S = 10 integers). It presents the performance of the three models of consistency (SC, DRC, and LRC) when considering two different read/write ratios (1/1 and 3/1).

From Figure 2 we can conclude that (i) when the ratio of read/write operations was 3/1 it was useful to use a replication protocol (write-invalidate or write-update) or the single-copy scheme (this is the case of write-mostly objects); (ii) when the number of reads is equal to the number of writes (ratio = 1/1) then the single-copy approach is more efficient; (ii) when the program uses small-size objects, the size of the update messages is also small and the difference between the two protocols (invalidate vs. update) is not quite visible.

The difference became more clear when we used large-size objects. As can be seen in Figure 3, the write-invalidation protocol (used for `READ_WRITE` objects) clearly outperformed the write-update protocol (`READ_MOSTLY` objects). The write-invalidation protocol is prone to cache-misses while the write-update protocol is not.

However, there is an important difference between the size of the invalidation messages and the messages used by the update protocol. This difference turns out to be relevant when the program uses large-size objects.

In this case, if the user decides to use the write-update protocol (i.e., READ-MOSTLY) then it is imperative to use the LRC model, because it is the only way to reduce the execution time closer to the write-invalidate protocol.

After comparing write-invalidate versus write-update, we want to state some conclusions about the performance of the models of consistency: DRC versus LRC. For the sake of normalization with other authors [27], we now use a mixed terminology: for instance. lazy-update means the system uses the LRC scheme and a write-update protocol of consistency, while eager-update means the DRC scheme was used with a write-update protocol.

From the data we collected in our system we observed the following:

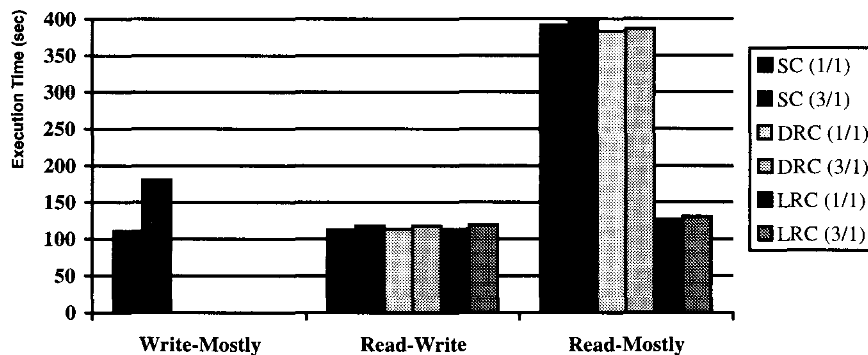


FIGURE 3 Performance results for large-size objects (S = 100K).

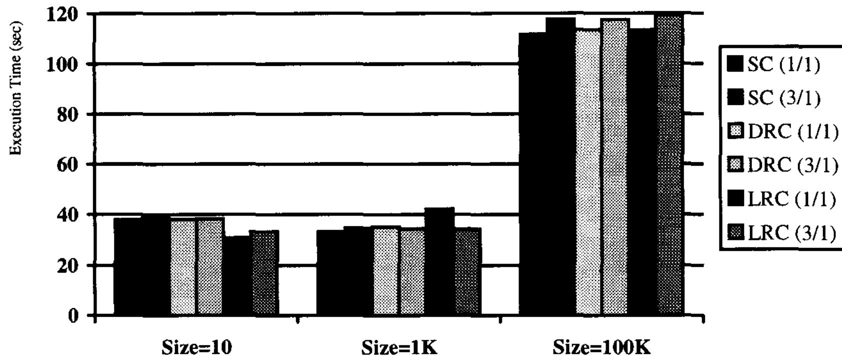


FIGURE 4 Performance of models of consistency for Read-Write objects.

1. Lazy-invalidate performs about the same as eager-invalidate.
2. Lazy-update clearly outperforms eager-update.
3. Eager-invalidate performs better than eager-update.
4. Lazy-update sends much less data than eager-update.

In Figure 4, we compare the three models of consistency for different sizes of the object, when using a write-in invalidate replication protocol. We can see by Figure 4 that lazy-invalidate (LRC) performs about the same as eager-update (DRC), independently of the size of the object.

The same does not happen when using a write-update protocol. The results are depicted in Figure 5, and we can see that in this case the performance difference between lazy-update and eager-update is quite clear. The difference becomes more evident when increasing the size of the shared object. While the LRC model effectively reduces the amount of data exchanged in update messages, the DRC scheme does not reduce the number of messages; it only tries to reduce the write latency, but it has been shown that this feature has almost no impact in the overall performance. If we compare Figures 4 and 5 we can further conclude that eager-invalidate performs better than eager-update.

Finally, we compare the amount of data exchanged by the DSM system, and in Figure 6 is shown a direct comparison among the three models when considering a ratio of 3/1 and large-size objects (S = 100K). The application executed for 10 iterations, and the amount of data is presented in Kilobytes. Figure 6 shows that a write-invalidate protocol exchanges less data than a single-copy approach or a write-update replication protocol. The only remarkable exception was observed for the lazy-update approach: We saw that LRC reduces the amount of data exchanged in update messages for about 80%.

Other similar experimental studies also present some related conclusions: For instance, [31] presents an LRC protocol for hardware coherent multiprocessors that outperforms an eager release consistency protocol (DASH protocol) by up to 17% on a variety of applications. It was also concluded that delaying the write notices until the acquire points is beneficial, but delaying their posting until a synchronization release

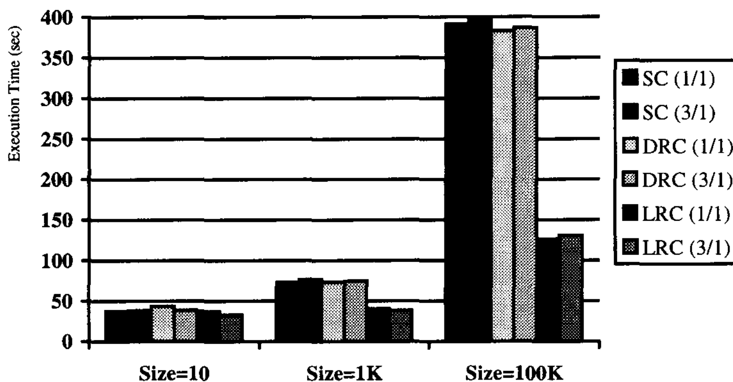


FIGURE 5 Performance of models of consistency for Read-Mostly objects.

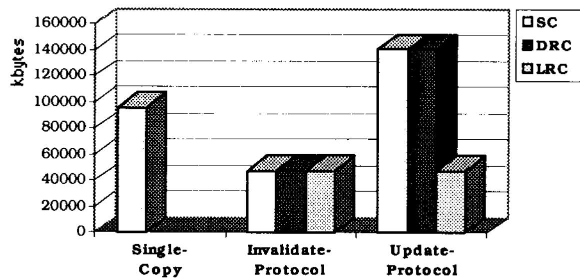


FIGURE 6 Amount of data exchanged (R = 3/1 and S = 100K).

point tends to move the coherence operations into the critical path of the application, resulting in even more synchronization overhead. This confirms our results.

Another interesting study presented in [27] also corroborates most of our conclusions with only a small difference: In their case, lazy-invalidate protocols outperform eager-invalidate protocols. This is not visible in our case. It probably has to do with some possible differences in the protocol implementations. However, in the overall, we all agree that lazy protocols can be beneficial: The basic advantage of the lazy protocols during lock transfers is that communication is limited to the two synchronizing processes. A release in an eager protocol often requires invalidate/update messages to be sent to several processes not involved in the synchronization transfer.

6.2 Comparing DSMPI with MPI Programs

In this subsection, we evaluate the overhead of using DSMPI against message passing. For that, we have implemented two versions of the following six applications:

TSP

TSP solves the traveling salesman problem using a branch-and-bound algorithm. The program follows the master/worker programming paradigm. The master generates the partial routes for the first three cities and places, them in a task queue that is a shared object globally accessed to the slave processes. After that step, the master behaves like a slave process and works on the problem cooperatively. Each slave grabs a task from that queue and computes all possible full routes based on those partial paths. There are two other shared objects which are replicated among the slave processes: One that keeps the minimum path and another that maintains the best route. The minimum value is frequently read and is used to prune part of the search tree. When a slaves finds a new solution it updates these two variables. The DSMPI version used locks to protect the access to the shared objects. We present results for the problem with 18 cities.

NQUEENS

NQUEENS solves the placement problem of N queens in an N-size chessboard. It also follows the master/worker programming model. Initially, one of the processes creates a bunch of tasks through the placement of the first two queens. Those tasks are placed in a task queue which is a shared data structure globally accessed to all the processes. The DSMPI version made use of locks to protect the task queue structure and other shared objects that keep the number of solutions achieved. In the network of workstations we executed the application for 13 queens, whereas for the Cray T3D we increased the problem to 14 queens.

SOR

Successive overrelaxation is an iterative method to solve Laplace's equation on a regular grid. The grid is partitioned into regions, each containing a band of rows of the global grid. Each region is assigned to a process. The update of the points in the grid is done by a red/black scheme. This requires two phases per iteration: One for black points and the other for red points. At the end of each iteration, the slave processes have to exchange the boundaries of their data blocks with two other neighbors. The DSMPI program uses a set of shared objects to exchange the row boundaries. At the end of each iteration, all the processes perform a global synchronization and evaluate a global test of convergence. This application used only barriers as synchronization variables. In the network of workstations, we executed the SOR application during 200 iterations in a grid of 1024×1024 (double precision), while in the Cray T3D the number of iterations was 500 and for the size of the grid 2048×2048.

GAUSS

GAUSS solves a system of linear equations using the method of Gauss elimination. The algorithm uses partial pivoting and distributes the columns of the input matrix among the processes in an interleaved way to avoid imbalance problems. At each iteration of the algorithm, one of the processes finds the pivot element and sends the pivot column to all the other processes. The DSMPI version used a shared object to store the pivot column and its index. A semaphore is used to protect the access to those shared objects. At the end of each iteration, all the processes have to execute a barrier synchronization. We executed the GAUSS application in a system of 2048 equations.

Table 1. Performance Results in a Network of Workstations

Application	MPI Version	DSMPI Version	DSMPI/MPI
TSP (18)	5 min 09 s 941	5 min 12 s 780	1.0091
NQUEENS (13)	2 min 57 s 124	2 min 59 s 558	1.0137
SOR (1024,2001)	6 min 21 s 555	6 min 44 s 043	1.0589
GAUSS (2048)	15 min 28 s 479	18 min 23 s 996	1.1890
ASP (1024)	10 min 03 s 562	10 min 37 s 376	1.0560
NBODY (4000)	10 min 05 s 652	10 min 06 s 342	1.0011

ASP

ASP solves the all-pairs shortest paths problem, i.e., it finds the length of the shortest path from any node i to any other node j in a given graph with N nodes using Floyd's algorithm. The distances between the nodes of the graph are represented in a matrix and each slave computes part of the matrix. It is an iterative algorithm, where in each iteration, one of the slaves keeps the pivot row. It broadcast its value by writing it to a shared object that is read by the other slaves. The DSMPI program uses locks to protect the access to that object and a barrier at the end of each iteration. In both environments, we executed the ASP application with a graph of 1024 nodes.

NBODY

This program simulates the evolution of a system of bodies under the influence of gravitational forces. Every body is modeled as a point mass that exerts forces on all other bodies in the system and the algorithm calculates the forces in a three-dimensional dimension. This computation is the kernel of particle simulation codes to simulate the gravitational forces between galaxies. We executed this application during 10 iterations for 4000 particles. The original algorithm used a ring structure for communication and at the end of each iteration it used a collective operation to calculate a minimum value among the processes. The DSMPI version makes use of semaphores to synchronize the communicating objects and a barrier at the end of each iteration.

First, we measured the performance in a network of workstations composed by four Sun4 machines connected by a 10 megabit/sec Ethernet. The overhead is represented in Table 1.

For some of the applications (TSP, NQUEENS, and NBODY), the overhead of the DSMPI version is almost negligible (0.91%, 1.37%, and 0.11%, respectively). The DSMPI version of SOR and ASP still has an acceptable overhead (5.89% and 5.60%), but the overhead for the GAUSS application was already considerable (18.90%). The reason for that difference is due to the extensive use of semaphores and large-size shared objects.

Table 2 presents the number of locks, semaphores, and barriers used by each application when executed on four processors. The GAUSS application was the application that used more synchronization mechanisms (semaphores and barriers). The ASP application was the second one on the list.

However, that number is not the only factor that interferes with the performance of DSMPI. Other important factors are the number of messages and the amount of data exchanged. Table 3 presents such values. In those figures, we did not take into account the data exchanged during the initialization part of the applications. The first column represents the number of read and write requests performed during the whole computation; the second column includes the amount of data sent in read or write requests; and finally the third column shows the amount of transmitted data per second.

We can see that the GAUSS application was the one that exchanged more data and sent the highest number of read/write requests. These two factors, together with the extensive use of semaphores and barriers, represent the reasons why the GAUSS application did not perform so well in DSMPI.

The application that transmitted more data per second was the SOR benchmark. Most of that data were sent in large-size messages.

At the end of the list there are the TSP, NQUEENS, and NBODY applications. These applications have a high computation to communication ratio: They per-

Table 2. Number of Synchronization Operations

Application	Locks	Semaphores	Barriers
TSP (18)	1124		2
NQUEENS (13)	536		2
SOR (1024,2001)			402
GAUSS (2048)		8162	2050
ASP (1024)		4096	1026
NBODY (4000)		240	12

Table 3. Amount of Data Exchanged in the DSMPI Version

Application	Number of Read+Write	Comm. Data (Kilobytes)	Kilobytes/s
TSP (18)	844	297	0.9
NQUEENS (13)	667	73	0.4
SOR (1024.200I)	4.800	38.400	95.0
GAUSS (2048)	16.384	65.600	59.4
ASP (1024)	4.096	16.384	25.7
NBODY (4000)	240	7.500	12.3

formed the smallest number of read and write requests and transmitted the smallest amount of data per second. Maybe this is why the DSMPI versions of those applications performed as well as the MPI counterparts. The results presented in Table 1 are similar to the ones obtained in the implementation of the Munin system [13]: Those authors achieved performance within 90–95% of the hand-coded message-passing implementations of the same application.

Another study presented in [32] compared the performance of TreadMarks [14] with PVM [33] and concluded that TreadMarks performed nearly identical to PVM for programs with high computation/communication ratio and large granularity of sharing. For programs with little access, locality and a large amount of shared data TreadMarks performed within 50% of PVM. For the remaining applications it performed within 75% of PVM.

The study presented in [27] shows that with current processors, the bandwidth of the 10 megabit/s Ethernet becomes a bottleneck, limiting the speedup and the efficiency of the applications. With 100 megabit/s ATM networks, which are now appearing on the market, those authors achieved considerable improvements in the performance of their DSM system.

To corroborate the idea that the bandwidth of the communication network is a feature of paramount importance, we conducted a similar experiment in the Cray T3D machine of the EPCC.* The results are shown in Table 4. The first two applications used eight processors: The DSMPI version used seven workers and one daemon process, whereas the MPI counterpart used seven workers and one master process. The remaining applications used 16 workers. The corresponding DSMPI version used four additional processors to run the daemon processes. Additional processors were required because the operating system (UNICOS) of the Cray only permits one process per processor.

Surprisingly, three of the applications (TSP, NQUEENS, and NBODY) performed better when using DSMPI than using MPI: 0.26%, 0.47%, and 16.53%, respectively. Reasons to explain such improvement include the replication of data through the local caches, the reduction of synchronization that exists in message passing, and the correct mapping of the shared objects. Nevertheless, as was seen previously, they represent the applications with the highest computation to communication ratio.

This set of results in the T3D suggest that when the underlying communication system is fast we can expect DSMPI to be directly competitive with message passing (at least for some applications). Table 5 presents the amount of data exchanged by the DSMPI versions, and if we compare these values with the ones presented in Table 3 we can observe that the communication throughput increased substantially in the Cray T3D. The amount of data transmitted is higher in the T3D because we executed the applications with more processors, as explained previously.

However, there is a point of concern in the DSMPI version that runs on the T3D which is the wasting of resources. As was said before, the UNICOS operating system only allows one process per processor and all the processors are loaded with the same binary. This implies that (i) each DSMPI daemon requires the exclusive use of one processor and (ii) the application code has to be gathered together with the daemon code in the same executable.

With such limitation, we can raise an obvious question: Assuming we have a fixed number of computing processors to be used by the whole application, what would be the optimum number of daemons?

Every DSMPI daemon is a data repository and is responsible for a certain number of application processes. This means that if we use only a small number of daemons for a large number of application processes, it may result in some daemon congestion. On

* Edinburgh Parallel Computing Centre, Scotland.

Table 4. Performance Results in the Cray T3D

Application	MPI Version	DSMPI Version	DSMPI/MPI
TSP	1 min 46 s 445	1 min 46 s 176	0.9974
NQUEENS (14)	7 min 43 s 863	7 min 41 s 685	0.9953
SOR (2048,100I)	1 min 57 s 803	2 min 02 s 323	1.0383
GAUSS (2048)	1 min 37 s 946	1 min 38 s 791	1.0086
ASP (1024)	1 min 38 s 068	1 min 44 s 369	1.0642
NBODY (4000)	52 s 498	43 s 821	0.8347

the other side, we know that if we decide to use more daemons we have to take some processors from the application.

We foresee that this decision is not very easy to take because it really depends on the characteristics of each application. However, to have a brief idea about the number of daemons we have to choose we have conducted an experiment using the NBODY application and a partition of 128 processors of the Cray T3D. The message-passing version made use of all processors, whereas the DSMPI version had to distribute the 128 processors between the application and the daemons. Figure 7 shows the measured execution time for different DSMPI configurations: The number of daemons goes from 2 to 28 (right to left in Figure 7). As can be seen, using only two or four daemons results in some congestion. Consequently, the DSMPI version performs worse than the MPI version. Using only two daemons really represents a clear bottleneck for the application. We had to increase the number of daemons and we found that the optimum number is 118 application processes and 10 daemons. In fact, some DSMPI configurations (120/8 till 116/12) presented a similar execution time to the MPI version that used all 128 processors for the application.

After that, we conducted another experiment with the NBODY application by running it on different partition sizes of the machine and comparing with the MPI counterpart. Table 6 shows the results of such an experiment. This time the number of simulated particles was 10,000. We executed the application in different partition sizes (8,16,32,64,128) and with a different number of DSMPI daemons: 1,2,2,4,10. As can be seen, DSMPI outperforms the MPI version of the NBODY program for most of the cases. Only when using 128 processors is there marginal overhead of the DSMPI version. We can also observe from Table 6 that this application scales very well with the increase in the number of processors.

7 CONCLUSIONS AND FUTURE WORK

DSMPI is a parallel library that supports the DSM abstraction in a portable way. It does not require any special feature from the operating system, uses a standard compiler, and assures full portability across all the platforms that support MPI. It is guaranteed that DSMPI programs can be executed in almost of the MPI-supported systems, without changing a line in the source code. The library provides different schemes of replication, and three different models of consistency. It has been shown that this feature offers increased flexibility to the DSM applications. The library also supports two additional but important features: fault-tolerance and heterogeneity.

Table 5. Amount of Data Exchanged in the DSMPI Version (Cray T3D)

Application	Number of Read+Write	Communication Data (Kilobytes)	Kilobytes/s
TSP (18)	855	601	5.6
NQUEENS (14)	790	204	0.4
SOR (2048,100I)	12,000	192,000	1569.6
GAUSS (2048)	98,304	393,792	3986.1
ASP (1024)	16,384	131,072	1255.8
NBODY (4000)	4,800	37,500	856.0

Table 6. Performance Results for Different Partition Sizes (NBODY 10.000)

Time MPI Version	Processes MPI/DSMPI	Time DSMPI Version	Difference
11 min 30 s 800	8w/7w	10 min 48 s 621	−6.10%
5 min 45 s 890	16w/14w	5 min 24 s 888	−6.07%
2 min 46 s 821	32w/30w	2 min 31 s 098	−9.42%
1 min 21 s 506	64w/60w	1 min 15 s 480	−7.39%
41 s 749	128w/118w	41 s 863	+0.27%

Heterogeneity is quite easy to support in a structure-based DSM system: The library knows about the format of each shared data object and it makes use of the heterogeneous support provided by MPI. In our opinion, the support for heterogeneity is one of the most appealing features of DSMPI.

DSMPI has some facilities for application checkpointing that can be very useful for the case of long-running computations. For more details about the mechanism, the interested reader is referred to [29].

However, the current version of DSMPI presents some drawbacks:

1. DSMPI does not automatically distribute the data (this is still the responsibility of the programmer). For the time being, although there is no automatic data distribution, we encourage programmers to use message passing during the initial data distribution phase, and then to use DSM for the rest of the computation.
2. DSMPI daemons are separate processes leading to a wasting of resources and some inefficiency, but this is a problem specific to the MPI implementations.
3. The current implementation uses static coherence protocols, i.e., the type of data objects remains fixed once specified. A self-adjusting scheme like the one presented in [24] would be an interesting scheme to provide.
4. The ownership protocol implemented is based on a static distributed scheme. The implementation of a dynamic distributed scheme and the direct comparison between both would give some interesting results.

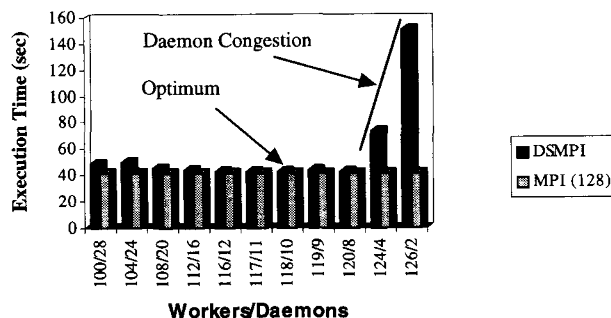


FIGURE 7 Changing the number of daemons for NBODY in 128 processors.

We plan to overcome some of these drawbacks in the next release that will be implemented on MPI-2. We look forward to a thread-safe implementation in order to redesign the DSMPI daemons that will be implemented as multithreaded servers. When possible we will use shared memory to communicate with the application process(es) that are located in the same host.

Despite these limitations, we consider that DSMPI is an easy-to-use and flexible library that presents an interesting level of efficiency and can be effectively used together with MPI. We do not support the idea that DSMPI is a replacement to MPI and message passing. DSMPI should be seen as a complement to MPI, allowing MPI programmers to share data within their applications. The possibility of integrating shared data and message passing is probably the most interesting feature of our system. The mixture of both programming models can be a very effective solution for parallel programming.

ACKNOWLEDGMENTS

This work herein presented was made when the first author was a visitor at EPCC. The visit was made possible due to the TRACS program. The first author thanks the TRACS team for their support, especially Andy Sanwell, Elspeth Minty, and Brian Fletcher. The first author is supported by JNICT on behalf of the "Programa Ciência" (BD-2083-92-IA). Finally, we thank the anonymous referees for their thoughtful comments in an earlier version of this article.

REFERENCES

[1] M. R. Eskicioglu, "A comprehensive bibliography of distributed shared memory," University of New Orleans, Tech. Rep. TR-95-01, May 1995.

[2] S. Raina, "Virtual shared memory: A survey of techniques and systems," University of Bristol, Tech Rep. CSTR-92-36, Dec. 1992.

[3] B. Nitzberg, V. Lo, "Distributed shared memory: A survey of issues and algorithms," IEEE Comput., vol. 24, pp. 52–60, August 1991.

[4] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Henessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. 17th Annual Int. Symp. Computer Architecture*, 1990, pp. 148–159.

[5] R. Bisiani and M. Ravishankar, "PLUS: A distributed shared-memory system," in *Proc. 17th Int. Symp. Computer Architecture*, 1990, pp. 115–124.

[6] J. Rothnie, "Overview of the KSR-1 computer system," Kendal Square Research, Waltham, MA, Tech. Rep. TR-9202001, 1992.

[7] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Systems*, vol. 7, pp. 321–359, Nov. 1989.

[8] B. Fleish and G. Popek, "Mirage: A coherent distributed shared memory design," in *Proc. 14th ACM Symp. Operating System Principles*, 1989, pp. 211–223.

[9] U. Ramachnadran and M. Y. Khalidi, "An implementation of distributed shared memory," *Software Practice Exp.*, vol. 21, pp. 443–464, May 1991.

[10] M. Kaashoek, H. Bal, and A. Tanenbaum, "Orca: A language for parallel programming of distributed systems," *IEEE Trans. Software Eng.*, vol. 18, pp. 190–205, March 1992.

[11] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," in *Proc. 12th ACM Symp. Operating System Principles*, Dec. 1989, pp. 147–152.

[12] R. Chandra, A. Gupta, and J. L. Hennessy, "COOL: An object-based language for parallel programming," *IEEE Comput.*, pp. 13–26, Aug. 1994.

[13] J. Carter, J. Bennet, and W. Zwaenepoel, "Implementation and performance of Munin," in *ACM Symp. Operating System Principles*, 1991, pp. 152–164.

[14] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed shared memory on standard workstations and operating systems," in *Proc. Winter 94 USENIX Conf.*, 1994, pp. 115–131.

[15] W. Y. Liang, "ADSMITH: A structure-based heterogeneous distributed shared memory on PVM," Institute of Computer Science National Tsing Hua University, Taiwan, Tech. Rep., June 1994.

[16] R. Bisiani and A. Forin, "Multilingual parallel programming of heterogeneous machines," *IEEE Trans. Comput.*, vol. 37, pp. 930–945, August 1988.

[17] S. Zhu, M. Stumm, K. Li, and D. Wortman, "Heterogeneous distributed shared memory," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, pp. 540–554, Sept. 1992.

[18] R. J. Harrison, "Moving beyond message passing: Experiments with a distributed-data model," Argonne National Laboratory, Argonne, IL, Tech. Rep., 1991.

[19] D. Kranz, K. Johnson, and A. Agarwal, "Integrating message-passing and shared memory: Early experience," in *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 1993, pp. 54–64.

[20] A. Cox, S. Dwarkadas, P. Keleher, and W. Zwaenepoel, "An integrated approach to distributed shared memory," in *Proc. 1st India Workshop on Parallel Computing*, 1994.

[21] MPI Forum, "A message passing interface standard," May 1994 (available on netlib).

[22] M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Comput.*, vol. 23, pp. 54–64, May 1990.

[23] H. Bal, M. F. Kaashoek, and A. Tanenbaum, "Replication techniques for speeding up parallel applications on distributed systems," *Concurrency Practice Exp.*, vol. 4, pp. 337–355, Aug. 1992.

[24] H. H. Wang and R. C. Chang, "A distributed shared memory system with self-adjusting coherence scheme," *Parallel Comput.*, vol. 20, pp. 1007–1025, 1994.

[25] H. Bal and M. F. Kaashoek, "Object distribution in Orca using compile-time and run-time heuristics," in *Proc. Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, 1993, pp. 162–177.

[26] P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proc. 19th Annual Int. Symp. Computer Architectures*, 1992, pp. 13–21.

[27] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel, "Evaluation of release consistent software distributed shared memory of emerging network technology," *ACM Comput. Architecture News*, vol. 21, pp. 144–155, May 1993.

[28] B. N. Bershad and M. J. Zekauskas, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," Carnegie-Mellon University, Pittsburgh, Tech. Rep. CMU-CS-91-170, Sept. 1991.

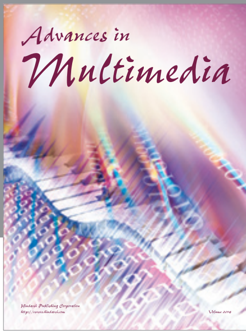
[29] L. M. Silva and J. G. Silva, "A checkpointing facility for an heterogeneous DSM system," Proc. ISCA International Conference on Parallel and Distributed Computing Systems, Dijon, France, pp. 554–559, Sept. 1996.

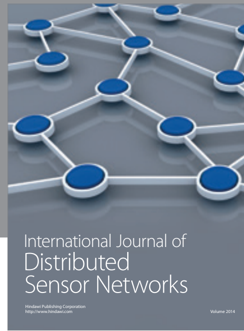
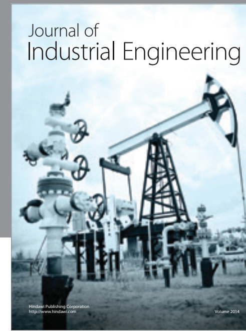
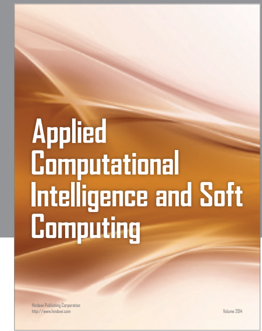
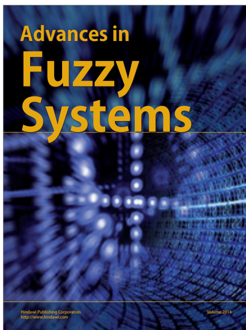
[30] S. Chandra, J. Larus, and A. Rogers, "Where is time spent in message-passing and shared-memory programs?", in *Proc. ACM ASPLOS VI*, 1994.

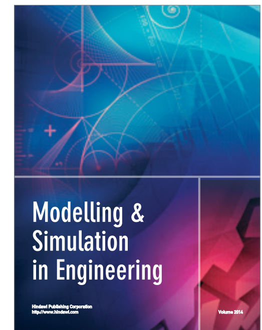
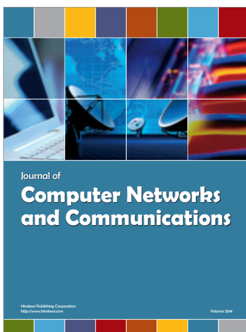
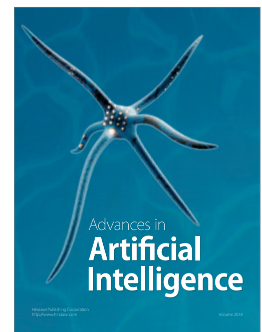
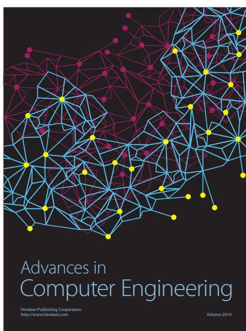
[31] L. I. Kontothanassis, M. L. Scott, and R. Bianchini, "Lazy release consistency for hardware-coherent multiprocessors," Department of Computer Science, University of Rochester, Rochester, NY, Tech. Rep., Dec. 1994.

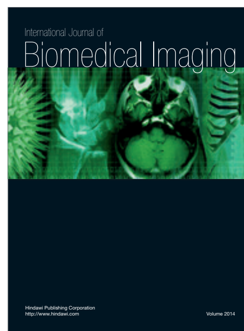
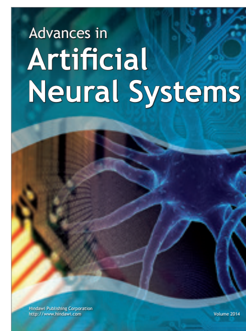
[32] H. Lu, "Message passing versus distributed shared memory on networks of workstations," Master Thesis, Department of Computer Science, Rice University, May 1995.

[33] G. A. Geist and V. S. Sunderam, "Network-based concurrent computing on the PVM system," *Concurrency Practice Exp.*, vol. 4, pp. 293–311, June 1992.

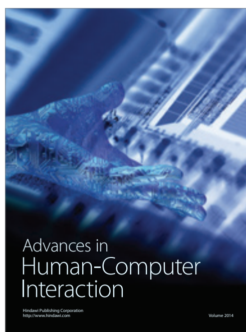
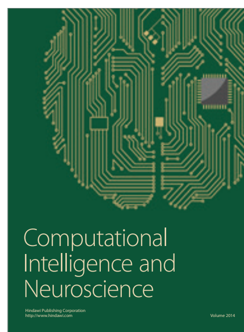

Advances in
Multimedia

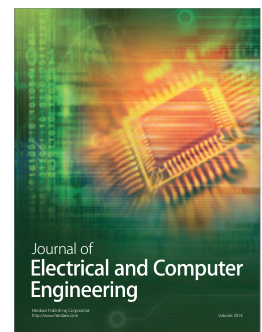
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering