EFFICIENT AND FLEXIBLE OBJECT SHARING

Miguel Castro Manuel Sequeira Manuel Costa Paulo Guedes
IST - INESC
R. Alves Redol 9, 1000 Lisboa, PORTUGAL
email: (miguel, mds, msc, pjg)@inesc.pt

Abstract – DiSOM is a software-based distributed shared memory (DSM) system, which supports intra- and inter-application sharing in heterogeneous networks of multiprocessor workstations. Unlike previous DSM systems, DiSOM provides fine-grained control over communication while retaining a simple shared memory model. It achieves this by using an update-based implementation of entry consistency, semaphores, remote object invocation, dynamic decomposition of objects and object-oriented language mechanisms. These techniques allow programmers to exploit application-specific knowledge to improve performance. A comparison between DiSOM and TreadMarks, a state-of-the-art DSM system, shows that on average DiSOM executes 33% faster, and sends 69% fewer messages and 38% less data.

INTRODUCTION

Distributed shared memory (DSM) systems [14, 10, 16, 9, 7, 4] offer the abstraction of a centralized memory that is shared by all the processors in a distributed system. This abstraction simplifies programming because it makes communication implicit. However, communication is frequently the main performance bottleneck in parallel applications, and the abstraction creates a barrier that prevents the programmer from taking advantage of application-specific knowledge to optimize communication.

DiSOM is a software-based DSM system, which supports intra- and inter-application sharing in heterogeneous networks of multiprocessor workstations. Unlike previous DSM systems, DiSOM provides fine-grained control over communication. It allows programmers to take advantage of application-specific knowledge to improve performance, and still benefit from the use of a simple shared memory programming model.

DiSOM offers a programming model where concurrent threads synchronize explicitly and communicate by executing operations on shared objects. The objects are kept consistent according to a variant of the *entry consistency* [3] memory model. This model requires shared objects to be explicitly associated with synchronization objects; and consistency is guaranteed as long as an access to a shared object is enclosed between an acquire and a release on a synchronization object associated with the shared object.

DiSOM implements entry consistency using an update protocol which piggy-backs all consistency messages on the synchronization protocol messages. Thus it ensures that communication occurs only when threads synchronize, allowing the programmer to control when to communicate.

When communication occurs, the implementation transfers only updates to the shared objects associated with the synchronization object. Therefore, the programmer can choose which data to transfer by controlling the association between shared data objects and synchronization objects. However, data objects are not always a convenient unit of synchronization and communication. DiSOM addresses this problem by providing a *region* mechanism that allows the programmer to dynamically associate regions of a data object with a synchronization object.

DiSOM uses object-oriented inheritance to allow programmers to customize system mechanisms and control communication at an even finer granularity. For example, each class can redefine the operations that are executed by the system when marshaling and unmarshaling its instances. The lower layers of the system are structured as a toolkit that ensures customization involves a minimum programming effort.

Barriers are a popular synchronization primitive in parallel programs. However, they typically over-synchronize cooperating threads and they can lead to the transmission of more data than needed. As an alternative, DiSOM offers an implementation of binary semaphores, optimized to support communication between one producer and several consumers in a distributed system. Semaphores provide the programmer with additional control over synchronization and communication. The performance results in the evaluation section show that semaphores can be used to improve performance over a barrier-based implementation.

DiSOM also offers a remote object invocation mechanism consistently integrated with the memory coherence protocol. This allows programmers to choose between function-shipping and data-shipping communication. This mechanism can be used to reduce the number of messages sent and the amount of data transferred, as shown in the evaluation section.

This paper compares the performance and programming complexity of DiSOM and TreadMarks [7], a state-of-the-art distributed shared memory system that implements lazy release consistency. The measurements were obtained by running the same set of applications both on DiSOM and on TreadMarks on the same cluster of workstations. The results show that on average DiSOM executes 33% faster, and sends 69% fewer messages and 38% less data. The improved performance comes at the expense of a slight increase in programming complexity.

The rest of the paper is organized as follows. First, we discuss related work. Next, we describe the programming model and its implementation. Later, we evaluate the performance and programmability of DiSOM.

RELATED WORK

The first software distributed shared memory systems [14, 10, 16, 9] used pages or coarse-grained segments as the unit of coherence, and had a sequential consistency[13] memory model.

A second generation of DSM systems explored relaxed memory consistency models to improve performance. Munin [4] provided a software implementation of release consistency [6] and TreadMarks introduced lazy release consistency [7]. In both these systems the unit of coherence is a virtual memory page and they support concurrent writers on the same coherence unit. TreadMarks uses an invalidate protocol: when a process executes an acquire it receives a notification with all the pages that have been modified and invalidates those pages. At page fault time, the process requests all the modifications made on the page, incorporates them on its copy of the page and unprotects the page. Processes maintain an original copy of each page they modify and compare the original copy to the modified page to determine which words were updated. Only those words are sent to a process requesting modifications to a page.

In TreadMarks, the programmer cannot control when communication occurs, because each cache miss involves communication; and cannot control what data is transferred, because the system automatically determines which data to transmit based on the access history. This causes the system to frequently send more messages and data than needed.

Midway [3] introduced the entry consistency memory model, implemented it using an update protocol, and made some preliminary comparisons with eager release consistency. This paper presents one of the first two comparison studies between an entry consistency based system and a lazy release consistency based system. The other study [8] compared TreadMarks with a system very similar to Midway, and concluded that there was no clear winner in terms of performance. Our study shows that DiSOM has significantly better performance than TreadMarks. The reason for this apparent contradiction is that, unlike Midway, DiSOM uses semaphores, remote object invocation and object-oriented language techniques to provide programmers with further control over communication.

Orca [2] and SAM [17] use objects as the unit of memory consistency like DiSOM. Orca uses compile time analysis to decide whether or not to replicate an object. Objects that are not replicated are accessed using remote procedure calls. Replicated objects have copies in all the processors in the system; read operations involve no communication; and write operations are broadcast to all the processors using a function shipping policy. DiSOM uses communication resources more efficiently and provides finer-grained control over communication. SAM allows programmers to choose between two consistency protocols optimized for different sharing patterns. Both of SAM's protocols can be emulated by DiSOM, and DiSOM's programming model is closer to the models offered by shared memory multiprocessors.

PROGRAMMING MODEL

The basic abstractions in DiSOM's programming model are objects and threads. Objects have a state composed of a set of fields and export a set of operations. Objects exist in an address space shared by all threads. Threads are active entities that synchronize explicitly and communicate by sharing objects. DiSOM offers three classes of synchronization objects, whose interfaces are summarized in table 1: the class EcObject implements read-write locks, the class EcBarrier implements barriers and the class EcSemaphore implements a variant of binary semaphores.

Table 1: Synchronization interfaces.

`EcObject::acqWrite()`	Acquire for writing
`EcObject::relWrite()`	Release from writing
`EcObject::acqRead()`	Aquire for reading
`EcObject::relRead()`	Release from reading
`EcBarrier::wait()`	Wait in barrier
`EcSemaphore::wait()`	Wait in semaphore
`EcSemaphore::signal()`	Signal semaphore
`EcSemaphore::enroll()`	Associate thread/semaphore
`EcSemaphore::unroll()`	Dissociate thread/semaphore

The objects in the EcObject class also define remote read and write operations. Executing a remote read operation on an object is semantically equivalent to forking and joining a thread that executes acqRead followed by the operation, and followed by relRead. A similar definition holds for remote write operations. These operations allow the programmer to access objects using a function-shipping policy instead of the usual data-shipping policy.

Each semaphore sem has a set of associated threads, $sem.threads$. In order to receive signal notifications, threads must associate themselves to the semaphore by calling `enroll`. When a thread signals sem, that signal is memorized by each thread in $sem.threads$ by setting a per-thread boolean variable. A `wait` operation completes when the boolean variable of the thread executing the operation is true, and it resets the boolean variable.

DiSOM uses the synchronization operations to drive the memory consistency protocol that implements the entry consistency [3] memory model. The memory consistency model is best described as a contract between the system and the application program. The contract specifies that programs must use synchronization operations to order all conflicting accesses (two accesses conflict if they are issued by different threads, access the same memory word, and at least one of them is a write). If a program satisfies this contract then the system guarantees that memory appears to be sequentially consistent [13].

The main difference between entry consistency and other weak consistency models, like weak and release consistency [1, 6, 12], is that in entry consistency each synchronization object s has a set of explicitly associated objects, $s.associated$, and the operations executed on s only order accesses to objects in $s.associated$. In other weak consistency models this relation is implicit, and synchronization operations order accesses to arbitrary objects. This restriction complicates programming but it can be used to reduce the amount of data that needs to be exchanged to ensure consistency; only the modified state of the objects in $s.associated$ needs to be exchanged. Another important difference is that, in order to update an object, a thread must acquire for writing a read-write lock associated with the object.

In DiSOM, all shared objects are instances of classes derived from EcObject and, therefore, have an implicitly associated read-write lock. The EcBarrier and EcSemaphore classes offer the methods `addObject` and `removeObject`, which allow the programmer to dynamically associate shared objects with barriers and semaphores.

The implicit association between a shared object and a synchronization object is a convenient model in an object-oriented programming environment because the object corresponds to a logical unit of data decomposition within the program, as opposed to a fixed size page. However, we frequently found that the object is ill-suited as the unit of synchronization because several portions of the object are independently shared. Since in our implementation the unit of synchronization is also the unit of communication, associating a synchronization object with a large object may also lead to the transmission of more data than required. The most common example of this problem is that of a large matrix where groups of lines or rows are independently shared. One solution to this problem is to force the programmer to statically decompose the matrix into objects that are independently shared [2, 17]. However, decomposing an object to achieve this goal frequently requires a significant programming effort, does not adapt to dynamic changes in the sharing patterns, and can lead to poor performance.

DiSOM handles the problem by allowing programmers to dynamically decompose an object into *regions*, i.e. subsets of fields. Regions are instances of EcRegion which is a subclass of EcObject. As for other objects, the system ensures consistency for regions provided the program synchronizes adequately, either using the region as a read-write lock or associated with other synchronization constructs.

The example in Figure 1 illustrates some important aspects of DiSOM's programming model. The program computes the product C of two matrices A and B, assigning a different stripe of C to each thread. A program in DiSOM starts its execution in function Main. In this case it creates the matrix multiplication application and forks a thread on each processor executing the application's method entry. The application's constructor creates the matrices and the synchronization objects, and associates each stripe of A with an element in stripesA. The method entry acquires the relevant parts of matrices A and B for reading; associates the stripe of matrix C to barrier done; and performs the multiplication.

The association between matrices A and C, and the corresponding synchronization objects is performed with a MatrixRegion object (MatrixRegion derives from EcRegion). The constructor of this class receives as arguments the matrix where the region is defined, the line where the region starts, and the number of elements in the region. It can optionally receive a stride.

IMPLEMENTATION

The programming model described in the previous section is implemented as a C++ class library that works on several UNIX variants. This section describes the implementation of the class library. It starts by discussing our approach to provide a shared address space and support heterogeneity. Then it describes the algorithm we use to implement the read-write locks that are

```
int Main(int argc, char **argv) {
    // allocate processors and multiply application
    Processors *p = new Processors(N);
    MatrixMultiply *app = new MatrixMultiply();

    app->start(p); // start threads

    // done, deallocate application and processors
    delete app;
    delete procs;
}

class MatrixMultiply : public Application {
    Matrix *A, *B, *C;
    EcBarrier *done;
    Array<MatrixRegion*> stripesA;
public:
    MatrixMultiply(void) : stripesA(N) {
        // create matrices and synchronization objects
        A = new Matrix(N, N);
        B = new Matrix(N, N);
        C = new Matrix(N, N);
        done = new EcBarrier(N);

        // associate the locks to the stripes of A
        for (i=0; i<N; i++)
            stripesA[i] = new MatrixRegion(A,i,N);
    }

    virtual int entry(int id) {
        // acquire A and B
        stripesA[id]->acqRead();
        B->acqRead();

        // associate the stripe of C to the barrier
        done->addObject(new MatrixRegion(C,id,N));

        // multiply the block
        for (int i=id; i<id+1; i++)
            for (int j=0; j<N; j++) {
                (*C)(i,j) = 0.0;
                for (int k=0; k<N; k++)
                    (*C)(i,j) += (*A)(i,k) * (*B)(k,j);
            }

        // release A and B
        B->relRead();
        stripesA[id]->relRead();

        // wait for the others and send the results
        done->wait();
    }
};
```

Figure 1: Programming example.

the basis of the EcObject class. The remote object invocation mechanism is integrated with the read-write lock algorithm and is described next. Later, we describe the implementation of the region mechanism, followed by a description of the implementation of semaphores and barriers. Finally, we discuss the hooks provided by DiSOM to allow programmers to customize system mechanisms.

The system is running on a cluster composed of several SPARCstation 10, i486 PC and DEC Alpha and a shared-memory multiprocessor Sun SPARCcenter 2000 with 10 processors. It provides source code portability across all platforms and takes advantage of the hardware shared memory on the SPARCcenter.

In the following description, the cluster is modeled as a set of processes with private memories which communicate exclusively by exchanging messages. Several threads may execute in each process. The communication channels between the processes are assumed to be reliable and FIFO. We assume no process failures in this paper. In a previous paper [15], we described a checkpointing protocol that allows DiSOM to support single failures efficiently.

Shared Address Space

Most distributed shared memory systems achieve uniform naming by mapping each shared data item to the same virtual memory address in each process. This technique cannot support a shared address space spanning heterogeneous architectures without incurring the overhead of address translation. Furthermore, existing systems that use this technique coordinate address space layout only within a single application and therefore fail to support inter-application sharing.

Since DiSOM supports sharing across different architectures, it uses a form of pointer swizzling [19] to achieve uniform object naming across different processes. The system assigns a global identifier to each shared object and automatically converts between the language level reference and the global identifier when the reference is exchanged in a message. DiSOM uses both the IP address of the machine where the process is running and the TCP/IP port of the process to make the object identifiers globally unique. Note that the identifiers will be globally unique even across different applications.

Our swizzling scheme is a form of *node marking* [11], i.e. when the memory consistency protocol brings an object's state into the address space of a process, all the references in the object are converted. Since the programming model requires explicit insertion of synchronization operations and explicit association of data to synchronization objects, those operations are used to detect non-resident objects, avoiding more expensive checking mechanisms. Our scheme is also a form of *direct swizzling*, i.e., global names are swizzled directly into pointers to objects. A context importing a given reference for the first time allocates space for the object and converts the global identifier into the pointer to the allocated copy. The copy is created using a class identifier that travels with the object's global identifier.

Heterogeneity

In order to perform data representation conversions when the state of an object is transferred between two processes of different architectures, the system must determine the type of the object at run-time. In DiSOM each object class defines a pack and an unpack method. These methods handle marshaling and unmarshaling of the object state and perform the needed conversions. This simple technique has been used by several object based distributed systems, for example Argus [5]. Conversions are performed using an *external data representation* policy that avoids the conversion when communicating processes have the same architecture.

Distributed Read-Write Locks

The distributed read-write lock algorithm implemented by the EcObject class is a variant of Li's dynamic distributed manager with distributed copy sets [14]. Li used this algorithm to keep memory coherent. We modified it to implement concurrent read/exclusive write synchronization of multiple threads running in multiple processes in a distributed system.

The algorithm associates two types of tokens with each lock, the *write token* and the *read token*. A process must hold the write token for one of its threads to perform an acqWrite. An acqRead will complete if the process holds a read token. The algorithm ensures that either exactly one process holds a write token and no read tokens exist; or one or more processes hold read tokens concurrently and no write token exists. This invariant enforces concurrent read exclusive write semantics.

Each lock has an associated *owner* that changes dynamically. The owner is either the process holding the write token or the last process to hold a write token. Each process maintains a *forwarding pointer* which points to the process it believes is the lock owner. A process also keeps a set of *token holders* that records the set of processes holding a read token received from that process.

Processes cache tokens and can re-acquire locks locally as long as their token is not invalidated. The algorithm sends a token request message when a thread calls acqWrite or acqRead and the enclosing process does not hold the needed token. The message is sent along the forwarding pointer chain and the thread blocks waiting for the reply. A write token can only be obtained from the owner process, but a read token can be obtained from any process holding a read token. Therefore, forwarding stops when the message reaches the owner or a process holding a read token. If the message reaches a process requesting a token for the same lock, forwarding is suspended until the local request is satisfied. This reduces the number of messages and provides better fairness guarantees.

When a process receives a request for a write token that it holds, it waits until the lock is released by all of its threads, and then it replies with a message transferring the write token and the token holders set to the requester. The owner then sets the forwarding pointer to point to the requester. When the requester receives the reply it merges the set of token holders in the reply with its own set. Then it sends messages to all the processes in the merged set of token holders, invalidating their read tokens, and waits for the replies. A process receiving an invalidate message will also send invalidate messages to the processes in its set of token holders. Thus invalidation of read tokens proceeds

in a distributed divide and conquer fashion. The processes that receive invalidation messages wait until local threads release the lock before replying. The invalidate message includes the identifier of the new owner that is used to set the forwarding pointer accordingly.

When a read token holder receives a request for a read token, the requesting process is inserted in the set of token holders, and a reply is sent to it. The reply includes the read token and the forwarding pointer. Thus all readers have their forwarding pointers set correctly. If the owner receives a read token request and it holds a write token, it proceeds as above but first converts its write token into a read token.

DiSOM implements entry consistency on top of the read-write locks using an update protocol that piggy-backs the object state in the token transfer messages. The read-write lock implementation calls the pack and unpack methods to marshal and unmarshal the object state into the token transfer messages. The update protocol provides programmers with a simple model, ensuring that communication only occurs at acquires.

Remote Object Invocation

The system supports synchronous remote read and write operations together with the algorithm described in the previous section. Thus it allows programmers to choose between function-shipping and data-shipping style communication. Remote operations can be used to reduce both the number of messages and the amount of data transferred as shown in the evaluation section.

The remote operations are implemented as a remote object invocation on top of the read-write lock protocol. When a remote operation is invoked on an object of the EcObject class, a message describing the operation and its arguments is sent along the forwarding pointer chain. In the case of a read operation, forwarding stops whenever a process with a read or write token is found, and the operation is executed by a thread in that process bracketed by an acqRead and a relRead pair. In the case of a write operation, forwarding stops only at the owner of the invoked object, and the operation is executed by a thread in that process bracketed by an acqWrite and a relWrite pair. The reply is sent directly to the invoking thread. This simple implementation ensures that remote operations always observe a consistent state and explores read replication for efficiency.

Regions

Regions allow programmers to decompose objects dynamically without impacting the performance of accesses to decomposed objects. They provide a level of indirection that allows programs to use the most efficient memory layout for their data. Regions allow programmers to choose a data layout independently of the parallelization model, the number of processors used, or how the work is distributed among processors.

Regions are instances of EcRegion which is a subclass of EcObject. The instance variables of regions are a reference to the decomposed object and some additional information describing the region. When the state of a region is sent in a message to another process, e.g. when an acquire is executed on the region, the additional information is passed to the decomposed object, which packs the portion of its state corresponding to the region into the message. Similarly, when the region is received in the remote process, the additional information is passed to the local replica of the decomposed object, that unpacks the data in the message into the memory locations corresponding to the abstract region.

Semaphores and Barriers

The semaphore implementation attempts to hide network latency by pushing new data asynchronously, and allowing programs to overlap communication and computation. It keeps a set with the identities of the associated threads and a boolean variable for each of the elements in the set. The boolean variable is kept in the process of the corresponding thread, so that it can be examined without communication. When a thread signals a semaphore, it sends the state of the objects associated with the semaphore to the associated threads in a single asynchronous message. A wait operation is allowed to complete only if the boolean variable associated with the thread executing the operation is set. This variable is set when a signal message is received, and it is reset when a wait operation performed by the thread completes.

The implementation of barriers uses a centralized master. The last thread in a process to cross a barrier sends an arrival message to the master. This message contains the objects associated with the barrier which are owned by the process along with their versions. The barrier master collects the objects in the arrival messages and ensures that the last versions prevail. The master sends a decision message to each slave after receiving messages from all slaves and after the last of the master's threads crosses the barrier. This message contains the last versions of the objects associated with the barrier.

The barrier implementation sends all the updates to all the threads crossing the barrier. However, these threads will not typically access all the objects associated with the barrier. Therefore, barriers will transfer more data than needed. On the other hand, the semaphore construct allows the programmer to specify which threads should receive the updates. Another advantage of semaphores is that they provide finer-grained synchronization than barriers. Barriers typically over-synchronize computing threads, but semaphores can be used to enforce synchronization constraints based only on data dependencies.

Customizing the System

One of the distinguishing features of DiSOM is the flexible fine-grained control it offers to applications. Classes can customize system actions for their particular instances using object-oriented inheritance. The lower layers of the system are organized as a toolkit that minimizes the programming effort required to customize system actions. For example, a class can override compiler-defined versions of the pack and unpack methods, that are used to marshal and unmarshal the state of the objects in the class.

A concrete example concerns the implementation of fine-grained access detection. DiSOM's memory model was defined to allow object-level access detection using the calls to the synchronization primitives. Therefore, there is no need to use page faults

or software dirty bits [20] to detect updates to shared objects. However, if objects are large and sparsely written (e.g. large arrays), this simple technique may transfer considerably more data than needed to ensure coherency. This happens because the update protocol conservatively transfers the entire object state to an acquiring thread. This problem is solved by providing a special array class that implements a software dirty bit scheme similar to Midway [20]. This class is a subclass of the regular array class. It redefines the pack and unpack methods to transfer only the updates, and uses an instrumented store operator to set the dirty bits.

EVALUATION

We evaluated DiSOM's performance and the complexity of its programming model by comparing it with TreadMarks [7]. The measurements were obtained using four shared memory parallel applications: Matrix Multiply (MM), Successive Over-Relaxation (SOR), Traveling Salesperson (TSP) and Water (Water). The code of SOR, TSP and Water was ported from TreadMarks.

The measurements were performed on a cluster of eight SPARCstations 10/30 connected by an otherwise idle 10 Mbps Ethernet network. Each SPARCstation 10/30 has a V8 SuperSPARC CPU running at 36 MHz, 36 Kbytes of internal cache and 32 Mbytes of memory. All the values presented are averages of three or more runs.

Applications

MM computes the product C of two matrices A and B of 512×512 single precision floating point numbers. C is statically divided in stripes with an equal number of rows and each stripe is assigned to a different thread. The result is collected by the initiating thread at the end of the computation. In DiSOM's implementation, the matrices are shared objects, and the stripes of matrices A and C are described by regions. The computing threads acquire all of B for reading, and their corresponding regions of A and C for reading and writing, respectively. The regions of C are associated with a barrier that is used to synchronize threads at the end of the computation. This barrier is the only synchronization used in TreadMark's implementation.

SOR is an implementation of the red-black successive over-relaxation algorithm [7]. SOR's input is a 1024×512 single precision floating point matrix and it performs 106 iterations. The matrix is statically divided in stripes with an equal number of rows and each stripe is assigned to a different thread. The interior elements of the matrix are initialized to small random values. Each iteration has two phases, one that computes the red elements and another that computes the black elements. Threads that were assigned neighboring stripes must exchange boundary elements of one color at the end of each phase. The result is collected by the initiating thread at the end of the computation.

This implementation of SOR places red and black elements in separate matrices. In TreadMarks, threads synchronize with a single barrier, crossed at the end of each phase. In DiSOM, the stripe assigned to a thread is represented by two regions, one in each matrix. These regions are associated with a barrier, used to synchronize threads at the end of the computation. Each thread has four associated semaphores, and each semaphore is associated with a region object describing a boundary line in one of the matrices. At the end of each phase, a thread signals the semaphores corresponding to the boundary regions just computed and waits in its neighbor's semaphores.

The TSP application solves the well known traveling salesperson problem in a graph of 19 cities, using a branch and bound algorithm [4]. The shared data structures in TSP are the minimum length tour encountered so far, an array of structures describing partially evaluated and unused tours, a priority queue with pointers to partially evaluated tours, and a stack with pointers to unused tours. Threads remove partial tours to evaluate from the priority queue and, if they are not long enough to apply an exhaustive search algorithm, compute new partial tours to be evaluated and insert them in the queue.

TreadMarks' implementation uses locks to synchronize accesses to the shared data structures. Similarly, in DiSOM, all the shared data structures are represented by shared objects and accesses are enclosed between the appropriate read-write lock synchronization operations. The array of tours and the priority queue are large objects that are sparsely modified between synchronization operations. Therefore, we used the version of the array class that implements fine-grained access detection.

Water is a N-body molecular dynamics simulation from the SPLASH benchmark suite [18]. It calculates forces and potentials in a system of water molecules in the liquid state. We measured 5 steps of a system with 343 molecules. The main data structures are an array of structures describing molecules and a set of global sums. Work is partitioned statically by assigning a set of contiguous molecules in the array to each thread, and having each thread compute the interactions between its molecules and the $343/2$ molecules that follow them in the array. We used the optimization suggested in [18] of collecting the changes to the molecules in private memory and updating the molecules only at the end of the inter-molecular forces calculation phase.

The TreadMarks' implementation uses a lock per molecule to synchronize updates, and uses barriers before and after the phases that compute inter-molecular interactions. In DiSOM, each molecule is a shared object and each thread has a semaphore to which it associates its molecules. Before the phases that compute inter-molecular interactions, each thread signals its semaphore and waits on the semaphores of half of the other threads. Remote write operations are used to update the molecules at the end of the phase that computes inter-molecular forces. Since only approximately $1/4$ of the object that describes a molecule is shared [18], DiSOM's implementation uses specialized versions of the pack and unpack methods that only transmit the shared fields.

Comparison between DiSOM and TreadMarks

The results presented in Figure 2 show the elapsed time to run the application, the total amount of data transferred and the total number of messages exchanged for the four applications. The application code is identical in both systems except for the different synchronization mechanisms as explained in the previous section.

DiSOM sends less data than TreadMarks in MM; with 8

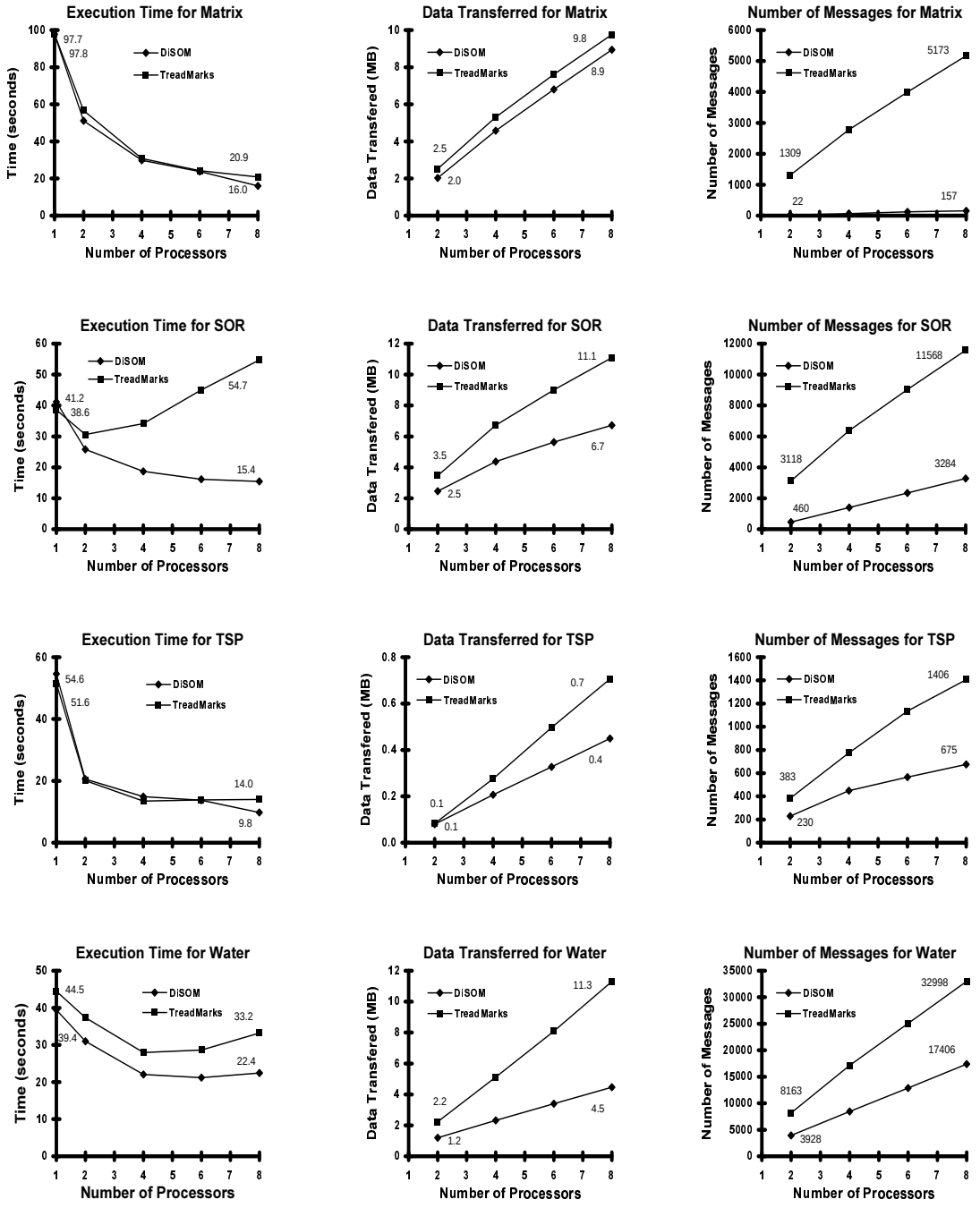

Figure 2: Execution times (seconds), total data transferred (MBytes) and number of messages exchanged for the four applications. The measurements were performed on a cluster of eight SPARCstations 10/30 connected by an otherwise idle 10 Mbps Ethernet network. All the values presented are averages of three or more runs.

processors, it sends 9% less data. The reason for the difference is that, in TreadMarks, threads are forced to acquire the initial values of their stripes of matrix C, and in DiSOM, they are not. DiSOM also sends much fewer messages; with 8 processors, it sends only 3% of the messages sent by TreadMarks. This happens because matrix B, and each stripe of matrices A and C are sent in a single message, whereas in TreadMarks they are fetched on demand page by page. The difference in the amount of data and number of messages exchanged leads to better performance in DiSOM. DiSOM's execution time with 8 processors is 23% less than TreadMark's.

In SOR, threads that were assigned neighboring stripes must exchange a minimum of 2 Kbytes, corresponding to the boundary regions, at the end of each phase. In DiSOM, only the minimum amount of data is exchanged. On the other hand, in TreadMarks, the threads also exchange the rest of the pages containing the boundaries, i.e approximately 4 Kbytes. Due to this false sharing problem, DiSOM sends significantly less data than TreadMarks; with 8 processors it sends 40% less data.

TreadMarks also sends approximately 3 times more messages than DiSOM per phase. DiSOM sends fewer messages because it uses semaphores with an update protocol. During each phase all but two threads (the one at the top and the one at the bottom of the matrix) signal 2 semaphores. The two threads at the boundaries of the matrix signal only one semaphore. Therefore, DiSOM sends $2(n-1)$ messages per phase, while TreadMarks sends $2(n-1)$ messages crossing the phase barrier plus $4(n-1)$ messages to fetch both boundary pages on demand. The overall difference is larger because the matrix stripes are acquired in two messages in DiSOM, and they are fetched on demand page by page in TreadMarks. Due to these differences, DiSOM performs much better than TreadMarks. DiSOM's smallest execution time is 49% less than TreadMarks' (30 seconds with 2 processors).

TreadMarks sends more data than DiSOM in TSP due to its fine-grained access detection technique. TreadMarks keeps updates in the form of differences between two versions of a page. When a process accesses an invalid page it must obtain all the differences created since the last time it accessed the page, and apply them in order to its old copy of the page. In this application, the stack and the priority queue have a migratory behavior forcing a process to acquire on average $n-1$ differences, where n is the number of processors, and there is a significant overlap between these differences. DiSOM also sends fewer messages than TreadMarks because it uses an update protocol that piggy-backs all coherency messages on the read-write lock token transfer messages. With 8 processors, DiSOM's execution time is 30% smaller than TreadMarks', because it sends 43% less data and 2.1 times fewer messages.

In Water, DiSOM sends less data than TreadMarks; with 8 processors, it sends 2.5 times less data, mainly because it uses specialized pack and unpack methods that only transfer the shared portion of a molecule object. Another reason is the use of remote object invocation. The specific contribution of each of these techniques is discussed in the next section. DiSOM also exchanges significantly fewer messages (1.9 times fewer messages with 8 processors) chiefly because it uses remote object invocations, but also due to the efficient update protocol. These differences lead to better performance, DiSOM's smallest execution time (20.1 with 6 processors) is 28% smaller than TreadMarks' (28.0 seconds with 4 processors).

The results presented above confirm the effectiveness of the techniques used in DiSOM to allow programmers to control communication. On average, DiSOM sends 69% fewer messages and 38% less data, leading to better overall performance.

Evaluation of Specific Techniques

This section discusses the impact of specific techniques on DiSOM's performance. It evaluates the impact of semaphores, remote object invocation and specialized pack and unpack methods, in the performance of Water.

We implemented a version of Water that uses a barrier, instead of semaphores, before the phases that compute intermolecular interactions. All the molecules are associated to this barrier. Figure 3 presents a comparison between the performance of this version and the version with semaphores.

The barrier implementation sends significantly more data than the one using semaphores (55% more with 8 processors). The reason is that when the barrier is crossed all the threads receive the state of all the molecules, whereas in the implementation with semaphores a thread receives the state of approximately half the molecules. Both versions exchange the same number of messages. The version with barriers is 18% slower than the version with semaphores but its performance is still better than TreadMarks'.

We performed a similar study for SOR and the version with barriers exchanged 3.8 times more data with 8 processors, leading to slow-down for more than 5 processors. Its performance was better than TreadMark's for a small number of processors but significantly worse for larger numbers of processors.

One solution to this problem is not to associate objects with barriers and instead acquire the objects for reading after the barrier crossing. This approach was used in [20]. It avoids sending more data than needed but it can send significantly more messages. For example, in Water with 8 processors it would send 2.5 times more messages than DiSOM and 1.3 times more messages than TreadMarks. Therefore, semaphores are a more efficient solution.

The versions of Water described so far use remote object invocation to update the molecule data directly in the process of the thread to which the molecule is assigned. This ensures that a thread is always the owner of its molecules, and therefore can always acquire them for writing without exchanging messages with other threads. Without remote operations, a thread must acquire the object describing the molecule and invalidate the object's copy in the remote process. This forces the thread to which the molecule is assigned to re-acquire the object. Therefore, a version of Water without remote object invocation sends more data and more messages.

The performance results presented in Figure 4 compare the version of Water with remote operations to a version without them, labeled "DiSOM no RemOps" in the figure. The results confirm the latter claim, the version without remote object invocation sends 38% more data and transfers 1.6 times more messages than the version with remote object invocation. These differences cause the version without remote invocations to perform significantly worse.

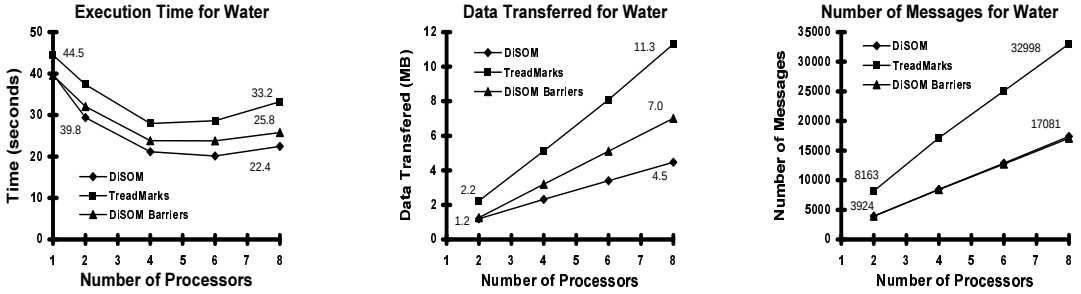

Figure 3: Comparison of semaphores vs. barriers for synchronization in Water.

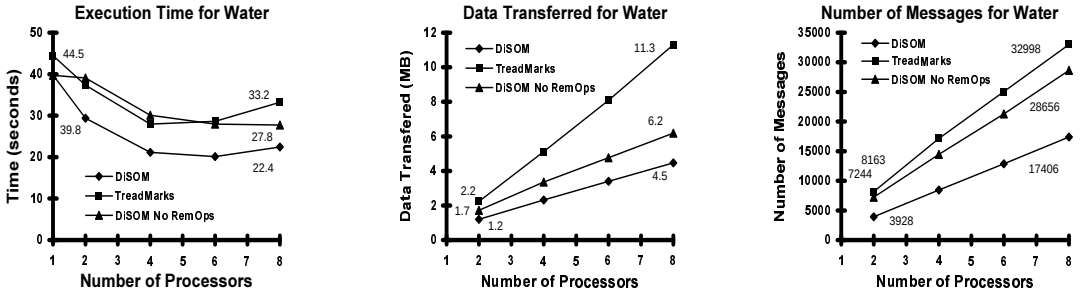

Figure 4: Effect of using remote operations in Water.

Our version of Water uses specialized pack and unpack methods in the class of objects that describe molecules. These methods only transfer the shared portion of the objects. Figure 5 compares the performance of Water with and without specialized pack and unpack methods. The version without specialized methods sends significantly more data (2.4 times more data with 8 processors) which leads to worse performance. Note that the version without specialized methods sends approximately the same amount of data as TreadMarks but fewer messages, however it has worse performance because TreadMarks' communication subsystem is more efficient than DiSOM's.

Like DiSOM, Midway uses an update-based implementation of entry consistency but it does not use the techniques described in this section. Therefore, the results presented in this section imply that an implementation of Water on Midway will have significantly worse performance than DiSOM's implementation.

Programming Complexity

Table 2 compares the code size of the applications in DiSOM and TreadMarks. The values were obtained by counting the number of semicolons in the code. The values obtained for DiSOM are larger than those obtained for TreadMarks due to a more complex programming model. Programming in DiSOM is more complex because the programmer has to explicitly associate data and synchronization objects, and in some places has to insert additional synchronization calls. The difference in code size is significant for the simple scientific kernels MM and SOR. However, for TSP the difference is 13% and for Water, the most realistic application, the difference is only 4%. These results support the claim that DiSOM's programming model is not significantly more complex than the one offered by TreadMarks.

Table 2: Application code size (obtained with `grep ';' *.[cChH] | wc`).

Application	DiSOM	TreadMarks
MM	206	162
SOR	163	91
TSP	386	340
Water	821	786

CONCLUSIONS

We presented a distributed shared memory system that provides programmers with fine-grained control over communication while retaining a convenient shared memory programming model. This is achieved by using an update-based implementation of entry consistency, efficient semaphores, remote object invocation, dynamic decomposition of objects, and allowing programmers to customize system actions on a per-object basis. Together these techniques provide a consistent framework for the development

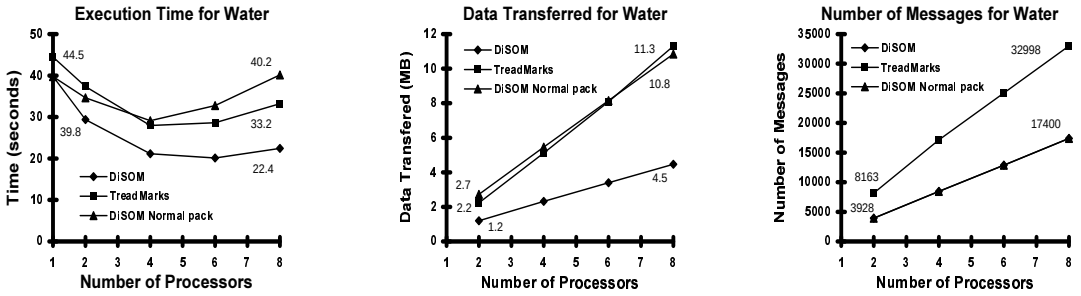

Figure 5: Effect of using specialized pack and unpack methods in Water.

of parallel and distributed applications.

The results of our comparative performance study show that DiSOM sends on average 69% fewer messages and 38% less data than TreadMarks, a state-of-the-art distributed shared memory system, at the expense of a slight increase in programming complexity. Therefore, the overall performance of DiSOM is significantly better, the smallest execution times on DiSOM are on the average 33% lower than TreadMarks'.

The increasing gap between processor performance and network latency stresses the need to offer fine-grained control over communication in order to achieve good performance. Therefore, the techniques used in DiSOM and its functionality make it an attractive system for the development of parallel and distributed applications in heterogeneous clusters of multiprocessor workstations.

ACKNOWLEDGEMENTS

We would like to thank Andrew C. Myers and the anonymous reviewers for their comments on earlier versions of this paper.

REFERENCES

[1] S. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE TPDS*, June 1993.

[2] H. Bal and A. Tanenbaum. Distributed Programming with Shared Data. In *IEEE Conference on Computer Languages*, 1988.

[3] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON*, February 1993.

[4] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *SOSP*, October 1991.

[5] B. Liskov et al. Implementation of Argus. In *SOSP*, November 1987.

[6] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, May 1990.

[7] P. Keleher et al. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Winter USENIX*, January 1994.

[8] S. Adve et al. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *IEEE HPCA*, February 1996.

[9] S. Zhou et al. Heretogeneous Distributed Shared Memory. *IEEE TPDS*, September 1992.

[10] B. Fleish and G. Popek. Mirage: A Coherent Distributed Virtual Memory Design. In *SOSP*, December 1989.

[11] A. Hosking and J. Moss. Protection Traps and Alternatives for Memory Management of an Object-Oriented Language. In *SOSP*, December 1993.

[12] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, May 1992.

[13] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE TOC*, September 1979.

[14] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM TOCS*, 7(4):321–359, November 1989.

[15] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *PODC*, August 1994.

[16] U. Ramachandran and M. Khalidi. An Implementation of Distributed Shared Memory. In *Distributed and Multiprocessor Systems Workshop*, October 1989.

[17] D. Scales and M. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *OSDI*, November 1994.

[18] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[19] P. Wilson and S. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *IWOOOS*, October 1992.

[20] M. Zekauskas, W. Sawdon, and B. Bershad. Software Write Detection for Distributed Shared Memory. In *OSDI*, November 1994.