Algorithms Implementing Distributed Shared Memory

Michael Stumm and Songnian Zhou
University of Toronto
Toronto, Canada M5S 1A4
Email: stumm@csri.toronto.edu

Abstract

A critical issue in the design of distributed applications is the appropriate choice of interprocess communication mechanism. Traditionally, mechanisms based on the *data passing model* are used, since they are a logical extension of the underlying communication system. However, mechanisms based on *distributed shared memory*, which provide distributed processes with a shared data address space accessible through *read* and *write* operations, have a number of advantages. In particular, the shared memory model provides an abstraction to the application programmer that is simpler and already well understood; for example, complex structures can be passed by reference.

This article categorizes and compares basic algorithms for implementing distributed shared memory and characterizes their performance. It is shown that, while the performance of distributed shared memory may be competitive with direct use of communication primitives, the algorithms are sensitive to the memory access behavior of the applications using this model. Based on simple analyses of data access costs, the essential characteristics of the algorithms are identified.

Keywords: Distributed Systems, Interprocess Communication, Distributed Shared Memory, Data Sharing, Data Consistency, Virtual Memory Systems, Performance Evaluation

1 Introduction

Traditionally, communication among processes in a distributed system is based on the *data passing* model. Message passing systems or systems that support remote procedure calls (RPC) adhere to this model. The data passing model is a logical and convenient extension to the underlying communication mechanism of the system; port or mailbox abstractions along with primitives such as `send` and `receive` are used for interprocess communication. This functionality can also be hidden in language-level constructs, as with RPC mechanisms. In either case, distributed processes pass shared information by *value*.

In contrast to the data passing model, the *shared memory* model provides processes in a system with a shared address space. Application programs can use this space in the same way they use normal local memory. That is, data in the shared space is accessed through `read` and `write` operations. As a result, applications can pass shared information by *reference*. The shared memory model is natural for distributed computations running on shared-memory multiprocessors. For loosely-coupled distributed systems, no physically shared memory is available to support such a model. However, a layer of software can be used to provide a shared memory abstraction to the applications. This software layer, which can be implemented either in an operating system kernel or, with proper system kernel support, in runtime library routines, uses the services of an underlying (message passing) communication system. The shared memory model applied to loosely coupled systems is referred to as *distributed shared memory.*

In this article, we describe and compare basic algorithms for implementing distributed shared memory by analyzing their performance. Conceptually, these algorithms extend local virtual address spaces to span multiple hosts connected by local area network, and some of them can easily be integrated with the hosts' virtual memory systems. In the remainder of this introduction, we describe the merits of distributed shared memory and the assumptions we make with respect to the environment in which the shared memory algorithms are executed. We then describe four basic algorithms are, provide a comprehensive analysis of their performance in relation to application-level access behavior, and show that the correct choice of algorithm is determined largely by the memory access behavior of the applications. We describe two particularly interesting extensions of the basic algorithms and conclude by observing some limitations of distributed shared memory.

Advantages of distributed shared memory

The primary advantage of distributed shared memory over data passing is the simpler abstraction provided to the application programmer, an abstraction the programmer already understands well. The access protocol used is consistent with the way sequential applications access data, allowing for a more natural transition from sequential to distributed applications. In principle, parallel and distributed computations written for a shared memory multiprocessor can be executed on a distributed shared memory system without change. The shared memory system hides the remote communication mechanism from the processes and allows complex structures to be passed by reference, substantially simplifying the programming of distributed applications.

In contrast, the message passing models force programmers to be conscious of data movement between processes at all times, since processes must explicitly use communication primitives and channels or ports. Also, since data in the data passing model is passed between multiple address spaces, it is difficult to pass complex data structures. Data structures passed between processes must be marshaled[1] and unmarshaled by the application.

For these reasons, the code of distributed applications written for distributed shared memory is usually significantly shorter and easier to understand than equivalent programs that use data passing.

The advantages of distributed shared memory have made it the focus of recent study and prompted the development of various algorithms for implementing the shared data model [3, 4, 11, 12, 7, 8, 9, 6]. Several implementations have demonstrated that, in terms of performance, distributed shared memory can compete with direct use of data passing in loosely coupled distributed systems[4, 11, 12, 13].In a few cases, applications using distributed shared memory can even outperform their message passing counterparts (even though the shared memory system is implemented on top of a message passing system). This is possible for three reasons.

(1) For shared memory algorithms that move data between hosts in large blocks, communication overhead is amortized over multiple memory accesses, reducing overall communication requirements if the application exhibits a sufficient degree of locality in its data accesses.

(2) Many (distributed) parallel applications execute in phases, where each compute phase is preceded by a data-exchange phase. The time needed for the data-exchange phase is often dictated

[1]Marshaling refers to the linearizing and packing of a data structure into a message.

by the throughput limitations of the communication system. Distributed shared memory algorithms typically move data on demand as they are being accessed, eliminating the data-exchange phase, spreading the communication load over a longer period of time and allowing for a greater degree of concurrency.

(3) The total amount of memory may be increased proportionally, reducing paging and swapping activity [12].

Similar systems

Distributed shared memory systems have goals similar to those of CPU cache memories in shared-memory multiprocessors, local memories in shared memory multiprocessors with non-uniform memory access (NUMA) times, distributed caching in network file systems, and distributed databases. In particular, they all attempt to minimize the access time to potentially shared data which is to be maintained consistent. Consequently, many of the algorithmic issues that must be addressed in these systems are similar.

Although these systems therefore often use algorithms that appear similar from a distance, their details and implementations can vary significantly because of differences in the cost parameters and in the ways they are used. For example, in NUMA multiprocessors, the memories are physically shared and time differential between accesses local and remote memory is lower than in distributed systems, as is the cost of transferring a block of data between the local memories of two processors. Hence, many of the algorithms we discuss in this article are of direct interest to designers of NUMA memory management systems, but the algorithms chosen to be most appropriate may differ.

Similarly, in bus-based shared memory multiprocessors, replication in the CPU caches (to avoid the much higher delays of accessing main memory and to reduce bus congestion) can be implemented cost-effectively because of the reliability and broadcast properties of the bus. On the other hand, in distributed systems where communication is unreliable, we show that algorithms without replication benefit certain types of applications. As a third example, distributed file systems and databases must provide for persistent data and, in the case of distributed databases, also reliability and atomicity. These requirements can significantly affect the choice of algorithm.

Model and environmental assumptions

In our discussions and analyses, we make certain assumptions with respect to the environment in which the algorithms are implemented. These are described here. The extrapolation of our results to other environments is left for future work.

In general, the performance of distributed and parallel applications is dictated primarily by communication costs, which in turn are dictated by the underlying hardware. Here, we assume a distributed system environment consisting of a cluster of hosts connected by a local area network, such as an Ethernet. In this environment, communication between processors is unreliable and slow relative to local memory access. We assume that broadcast and multicast communication, where a single message can be sent (unreliably) to multiple receivers in a single network transaction, is available. Most bus and ring networks fit this description.

For performance analysis, communication costs are abstracted in terms of the number of *messages* sent and the number of *packet events*. A packet event is the cost associated with either receiving or sending a small packet (about 1 ms on a SUN-3/50). A point-to-point message transfer therefore requires one message, a packet event at the sending site and a packet event at the receiving site. A multicast or broadcast message transmission requires one message, a packet event at the sending site and a packet event at each receiving site.

The shared memory model provides two basic operations for accessing shared data:

```
data := read( address )
write( data, address )
```

`Read` returns the *data item* referenced by `address`, and `write` sets the contents referenced by `address` to the value of `data`. For simplicity, the algorithms for implementing shared data are described in terms of these two operations. We assume that the distributed applications call these functions explicitly, although this may not always be necessary with suitable compiler and/or operating system support, and that the data item accessed is always a single word.

Of course, variations in the syntax and semantics of these operations are possible. For instance, the operations may be called by a number of different names, such as fetch/store, in/out, etc. The type of the data being accessed can also vary and include for example integers, byte arrays, or more complex user-defined types. Finally, the semantics of the memory access functions can go beyond those offered by traditional memory systems and can include atomic enqueueing or dequeueing operations or even entire database operations. For example, Linda [1] is a programming language that directly supports a shared data space by providing a shared tuple space and three basic operations: `read` reads an existing data item called "tuple" from the tuple space, `out` adds a new tuple and `in` reads and then removes an existing tuple. Linda's tuples are addressed by content rather than by location.

2 Basic algorithms

This section describes four basic distributed shared memory algorithms. For each of these algorithms we consider the cost of read and write operations and issues in their implementation. The algorithms described can be categorized by whether they migrate and/or replicate data, as depicted in Figure 1. Two of the algorithms migrate data to the site where it is being accessed in an attempt to exploit locality in data accesses and decrease the number of remote accesses, thus avoiding communication overhead; and two of the algorithms replicate data so that multiple read accesses can take place at the same time using local accesses.

Implementations of distributed shared memory based on replication should make this replication transparent to the applications. In other words, processes should not be able to observe (by reading and writing shared data) that all data accesses are not directed to the same copy of data.

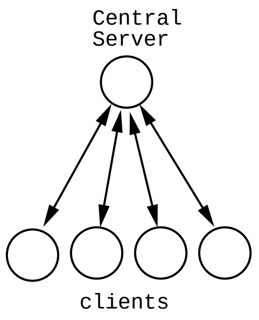
More formally [10], the result of applications using shared data should be the same as if the memory operations of all hosts were executing in some sequential order, and the operations of each individual host appear in sequence in the order specified by its program, in which case the shared memory is said to be *consistent*. Shared memory in a shared-memory multiprocessor is expected to behave this way.

This definition of consistency should not be confused with a stricter definition requiring read accesses to return the value of the most recent write to the same location, which is naturally applicable to concurrent processes running on a uniprocessor but not necessarily to those on shared-memory multiprocessors with CPU caches and writeback buffers [5].[2] If the stricter definition holds then so does the weaker definition (but the converse is not true).

[2]Our sense of consistency is also different from serializability in databases. We consider each read or write operation to be independent of one another.

	non-replicated	replicated
non-migrating	**Central Server**	**Full-Replication**
migrating	**Migration**	**Read-Replication**

Figure 1: Four distributed memory algorithms



Client	Central Server
send data request	
	receive request perform data access send response
receive response	

Figure 2: The central-server algorithm

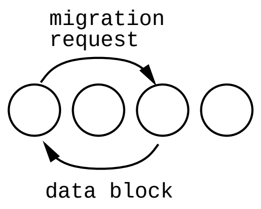
2.1 The central-server algorithm

The simplest strategy for implementing distributed shared memory uses a central server that is responsible for servicing all accesses to shared data and maintains the only copy of the shared data. Both read and write operations involve the sending of a request message to the data server by the process executing the operation, as depicted in Figure 2. The data server executes the request and responds either with the data item in the case of a read operation or with an acknowledgment in the case of a write operation.

A simple request-response protocol can be used for communication in implementations of this algorithm. For reliability, a request is retransmitted after each timeout period with no response. This is sufficient, since the read request is idempotent; for write requests, the server must keep a sequence number for each client so that it can detect duplicate transmissions and acknowledge them appropriately. A failure condition is raised after several timeout periods with no response.

Hence, this algorithm requires two messages for each data access: one from the process requesting the access to the data server and the other containing the data server's response. Moreover, each data access requires four packet events: two at the requesting process (one to send the request and one to receive the response) and two at the server.

One potential problem with the central server is that it may become a bottleneck, since it has to service the requests from all clients. To distribute the server load, the shared data can be distributed onto several servers. In that case, clients must be able to locate the correct server for data access. A client can multicast its access requests to all servers, but this would significantly increase the load on all servers, since every server would incur the overhead of a packet event for each such request. A better solution is to partition the data by address and use some simple mapping function to decide which server to contact.



Client	Remote Host

if block not local
  determine location
  send request	

	receive request send block
receive response access data	

Figure 3: The migration algorithm

2.2 The migration algorithm

In the migration algorithm, depicted in Figure 3, the data is always migrated to the site where it is accessed. This is a “single reader / single writer” (SRSW) protocol, since only the threads executing on one host can be reading or writing a given data item at any one time. Instead of migrating individual data items, data is typically migrated between servers in a fixed size unit called a *block* to facilitate the management of the data. The advantage of this algorithm is that no communication costs are incurred when a process accesses data currently held locally.

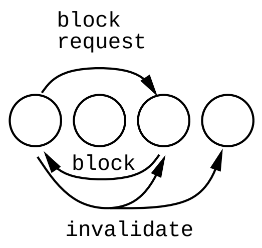
If an application exhibits high locality of reference, the cost of data migration is amortized over multiple accesses. However, with this algorithm, it is also possible for pages to *thrash* between hosts, resulting in few memory accesses between migrations and thereby poor performance. Often, the application writer will be able to control thrashing by judiciously assigning data to blocks.

A second advantage of the migration algorithm is that it can be integrated with the virtual memory system of the host operating system if the size of the block is chosen to be equal to the size of a virtual memory page (or a multiple thereof). If a shared memory page is held locally, it can be mapped into the application’s virtual address space and accessed using the normal machine instructions for accessing memory. An access to a data item located in data blocks not held locally triggers a page fault so that the fault handler can communicate with the remote hosts to obtain the data block before mapping it into the application’s address space. When a data block is migrated away, it is removed from any local address space it has been mapped into.

The location of a remote data block can be found by multicasting a migration request message to all remote hosts, but more efficient methods are known [11]. For example, one can statically assign each data block to a managing server that always knows the location of the data block. To distribute the load, the management of all data blocks is partitioned across all hosts. A client queries the managing server of a data block both to determine the current location of the data block and to inform the manager that it will migrate the data block.

2.3 The read-replication algorithm

One disadvantage of the algorithms described so far is that only the threads on one host can access data contained in the same block at any given time. Replication can reduce the *average* cost of read operations, since it allows read operations to be simultaneously executed locally (with no communication overhead) at multiple hosts. However, some of the write operations may become more expensive, since the replicas may have to be invalidated or updated, in order to maintain consistency. Nevertheless, if the ratio of reads over writes is large, then the extra expense for the write operations may be more than offset by the lower average cost of the read operations.



```
Client                         | Remote Host
-------------------------------+--------------------
if block not local             |
   determine location          |
   send request                |
                               | receive request
                               | send block
   receive block               |
multicast invalidate           |
                               | receive invalidate
                               | invalidate block
access data                    |
```

Figure 4: The write operation in the read-replication algorithm

Replication can be naturally added to the migration algorithm by allowing either one site a readable/writable copy of a particular block, or multiple sites read-only copies of that block. This type of replication is referred to as "multiple readers / single writer" (MRSW) replication.

For a read operation on a data item in a block that is currently not local, it is necessary to communicate with remote sites to first acquire a read-only copy of that block and to change to read-only the access rights to any writable copy if necessary before the read operation can complete. For a write operation to data in a block that is either not local or for which the local host has no write permission, all copies of the same block held at other sites must be invalidated before the write can proceed. (See Figure 4.)

This strategy resembles the *write invalidate* algorithm for cache consistency implemented by hardware in some multiprocessors [2]. The read-replication algorithm is consistent, because a read access always returns the value of the most recent write to the same location.

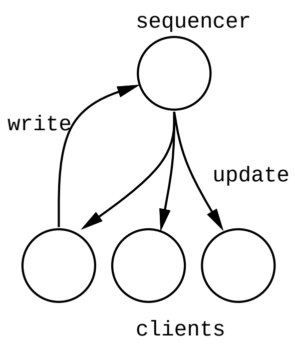
This type of replication has been investigated extensively by Li [11, 12]. In Li's implementation, each block has a server designated as its *owner*, that is responsible for maintaining a list of the servers having a read-only copy of that block. This list is called the block's *copy-set*.

A read (or write) access to a block for which a server does not have the appropriate access rights causes a read (or write) fault. The fault handler transmits a request to the server that has ownership of the appropriate block. For a read fault, the owning server replies with a copy of the block, adds the requesting server to the copy-set of the requested block, and changes the access rights of its local copy to read-only, if necessary.

When a write fault occurs, ownership for the block is transferred from the previous owner to the server where the write fault occurred. After receiving the response, the write-fault handler requests all servers in the copy-set to invalidate their local copy, after which the access rights to that block are set to write access at the site of the new owner and the copy-set is cleared.

2.4 The full-replication algorithm

Full replication allows data blocks to be replicated even while being written to. This algorithm therefore adheres to a "multiple readers / multiple writers" (MRMW) protocol. Keeping the data copies consistent is straightforward for non-replicated algorithms, since accesses to data are sequenced according to the order in which they occur at the site where the data is held. In the case of fully replicated data, accesses to the data must either be properly sequenced or controlled in order to ensure consistency.



Client	Sequencer	Hosts
if write		
send data		
	receive data	
	add seq. no.	
	multicast	
receive ack		receive data
update local		update local
memory		memory

Figure 5: The full-replication algorithm

One possible way to keep the replicated data consistent is to *globally* sequence the write operations, while only sequencing the read operations relative to the writes that occur local to the site where the reads are executed. For example, the *write update* algorithm for cache consistency implemented by hardware in some multiprocessors [2] maintains consistency in this fashion, that is, reads occur locally from the cache while writes are broadcast over the bus that sequences them automatically.

A simple strategy based on sequencing uses a single global gap-free sequencer, which is a process executing on a host participating in distributed shared memory. When a process attempts a write to shared memory, the intended modification is sent to the sequencer. This sequencer assigns the next sequence number to the modification and multicasts the modification with this sequence number to all sites.

Each site processes broadcast write operations in sequence number order. When a write arrives at a site, the sequence number is verified to be the next expected one. If a gap in the sequence numbers is detected, either a modification was missed or a modification was received out of order, in which case a retransmission of the modification message is requested[3]. In effect, this strategy implements a negative acknowledgment protocol.

In the common case within our assumed environment, packets arrive at all sites in their proper order. Therefore, a write requires two packet events at the writing process, two packet events at the sequencer, and a packet event at each of the other replica sites, for a system total of $S + 2$ packet events with S participating sites.

Many variants to the above algorithms exist. For example, Bisiani and Forin [3] describe an algorithm for full replication, which uses the same principle as the sequencing algorithm to ensure individual data structures remain consistent. However, rather than using a single server to sequence all writes, writes to any particular data structure are sequenced by the server which manages the *master copy* of that data structure. Although each data structure is maintained in a consistent manner, there is no assurance with this algorithm that updates to multiple data structures are made consistently.

3 Performance comparisons

All of the four algorithms described in the previous section ensure consistency in distributed shared memory. However, their performance is sensitive to the data access behavior of the application. In

[3]This implies that somewhere a log of recent write requests must be maintained.

p:	The cost of a packet event, i.e., the processing cost of sending or receiving a short packet, which includes possible context switching, data copying, and interrupt handling overhead. Typical values for real systems range from one to several milliseconds.
P:	The cost of sending or receiving a data block. This is similar to p, except that P is typically significantly higher. For an 8 KB block, where often multiple packets are needed, typical values range from 20 to 40 ms. For our analyses, only the ratio between P and p are important, rather than their absolute values.
S:	The number of sites participating in distributed shared memory.
r:	Read/write ratio, i.e., there is one write operation for every r reads on average. This parameter is also used to refer to the access pattern of entire blocks. Although the two ratios may be different, we assume they are equal in order to simplify our analyses.
f:	Probability of an access fault on a *non-replicated* data block (used in the migration algorithm). This is equal to the inverse of the average number of consecutive accesses to a block by a single site, before another site makes an access to the same block, causing a fault. f characterizes the locality of data accesses for the migration algorithm.
f':	Probability of an access fault on *replicated* data blocks used in the read-replication algorithm. It is the inverse of the average number of consecutive accesses to data items in blocks kept locally, before a data item in a block not kept locally is accessed. f' characterizes the locality of data accesses for the read-replication algorithm.

Table 1: Parameters that characterize the basic costs of accessing shared data.

this section, we identify the factors in data access costs and investigate the application behaviors that have significant bearings on the performance of the algorithms. Based on some simple analyses, we compare the relative merits of the algorithms in an attempt to unveil the underlying relationship between access patterns of applications and the shared memory algorithms that are likely to produce better performance for them.

3.1 Model and assumptions

The parameters in Table 1 are used to characterize the basic costs of accessing shared data and the application behaviors. Among them, the two types of access fault rates, f and f', have the greatest impact on performance on the corresponding algorithms, but, unfortunately, are also the most difficult to assess, since they vary widely from application to application. It should also be pointed out that these parameters are not entirely independent of one another. For instance, the size of a data block and therefore the block transfer cost, P, influences f and f', in conflicting directions. As the block size increases, more accesses to the block are possible before another block is accessed; however, access interferences between sites become more likely. S also has direct impact on the fault rates. Nevertheless, the analyses below suffice to characterize the shared memory algorithms.

To focus on the essential performance characteristics of the algorithms and to simplify our analyses, a number of assumptions are made:

1. The amount of message traffic will not cause network congestion. Hence, we will only consider the message processing costs, p and P, but not the network bandwidth occupied by messages.

2. Server congestion is not serious enough to cause significant delay in remote access. This is reasonable for the algorithms we study, since there are effective ways to reduce the load on the servers (see Section 2.1).

3. The cost of accessing a locally available data item is negligible compared to remote access cost. They therefore do not show up in our access cost calculations below.

4. Message passing is assumed to be reliable, so the cost of retransmission is not incurred. Note, however, that the cost for acknowledgment messages, required to determine whether a retransmission is necessary or not, is included in our models.

To compare the performance of the distributed shared memory algorithms, we need to define a performance measure. Distributed shared memory is often used to support parallel applications in which multiple threads of execution may be in progress on a number of sites. We therefore choose the *average cost per data access* to the *entire* system as the performance measure. Hence, if a data access involves one or more remote sites, the message processing costs on both the local and remote site(s) are included.

Using the basic parameters and the simplifying assumptions described above, the average access costs of the four algorithms may be expressed as follows:

$$\begin{aligned}
\text{Central server:} \quad C_c &= (1 - \frac{1}{S}) * 4p \\
\text{Migration:} \quad C_m &= f * (2P + 4p) \\
\text{Read replication:} \quad C_{rr} &= f' * [2P + 4p + \frac{Sp}{r+1}] \\
\text{Full replication:} \quad C_{fr} &= \frac{1}{r+1} * (S + 2)p
\end{aligned}$$

Each of these expressions has two components. The first component, to the left of the '$*$', is the probability of an access to a data item being remote. The second component, to the right of the '$*$', is equal to the average cost of accessing a remote data item. Since the cost of local accesses is assumed to be negligible, the average cost of accessing a data item is therefore equal to the product of these two components.

In the case of the central server algorithm, the probability of accessing a remote data item is $1 - \frac{1}{S}$, in which case 4 packet events are necessary for the access (assuming that data is uniformly distributed over all sites). The overall cost, C_c, is therefore mainly determined by the cost of a packet event, as long as the number of sites is over 4 or 5.

For the migration algorithm, f represents the probability of an accessing a non-local data item. The cost of accessing a data item in that case is equal to the cost of bringing the data block containing the data item to the local site, which includes a total of one block transfer ($2P$) and four packet events distributed across the local, manager and server sites. We assume that the local, manager, and server sites are all distinct, and that the request is forwarded by the manager to the server. The sequence of packet events are send (on local site), receive (on manager site), forward (on manager site), and receive (on server site).

For read replication, the remote access cost approximates that of the migration algorithm, except that, in the case of a write fault (which occurs with a probability of $\frac{1}{r+1}$), a multicast invalidation packet must be handled by all S sites. The block transfer cost is always included in our expression, although it may not be necessary if a write fault occurs and a local (read) copy of the block is available.

Finally, for the full-replication algorithm, the probability of a remote access is equal to the probability of a write access The associated cost for this write is always a message from the local site to the sequencer (2 packet events), followed by a multicast update message to all other sites (S packet events).

3.2 Comparative analyses

The above discussion prepares us to make some *pair-wise* comparisons of the algorithms' performance to illustrate the conditions under which one algorithm may outperform another. Each comparison is made by equating the average costs of the two algorithms concerned, to derive a curve along which they yield similar performance.

This curve, which we call the *equal-performing curve*, divides the parameter space into two halves such that in each half one of the algorithms will perform better than the other. For example, in the following comparison between migration and read replication, the equation on the right of Figure 6 is that of the equal-performing curve, derived from $C_m = C_{rr}$ (with some rearrangement). Since all of the cost formulas include packet cost p, only the ratio between P and p matters in the following comparative analyses. The value of P/p is assumed to be 20. Based on these comparisons, we will be able to make some general comments on performance.

Migration vs. read replication

The only difference between these two algorithms is that replication is used in the read-replication algorithm to allow interleaved reads on several sites without block movements, but at the expense of multicasting invalidation requests upon updates. Interestingly, as shown in Figure 6, the invalidation traffic does not have a strong influence on the algorithms' relative performance. As long as the cost of a block transfer is substantially higher than that of a small message, the curves for different values of S and r cluster closely together and are very close to the $f = f'$ line.

Typically, read replication effectively reduces the block fault rate, because, in contrast to the migration algorithm, interleaved read accesses to the same block will no longer cause faults, so the value of f' is smaller than f. Therefore, one can expect read replication to outperform migration for a vast majority of applications.

Central server vs. read replication

Figure 7 compares the central-server and read-replication algorithms. The equal-performing curve is almost flat, that is, insensitive to the number of sites. Moreover, the influence of the read/write ratio is also minimal. Hence, the key consideration in choosing between the two algorithms is the locality of access. Typically, a block fault rate of 0.07 (14 accesses between faults) is considered very high (faults very frequent). Therefore, Read replication appears to be more favorable for many applications.

Read replication vs. full replication

Both algorithms use read replication. The full-replication algorithm is more aggressive in that multiple copies are maintained even for updates. Figure 8 shows that the relative performance

$$f = f'[1 + \frac{S}{44(r+1)}]$$

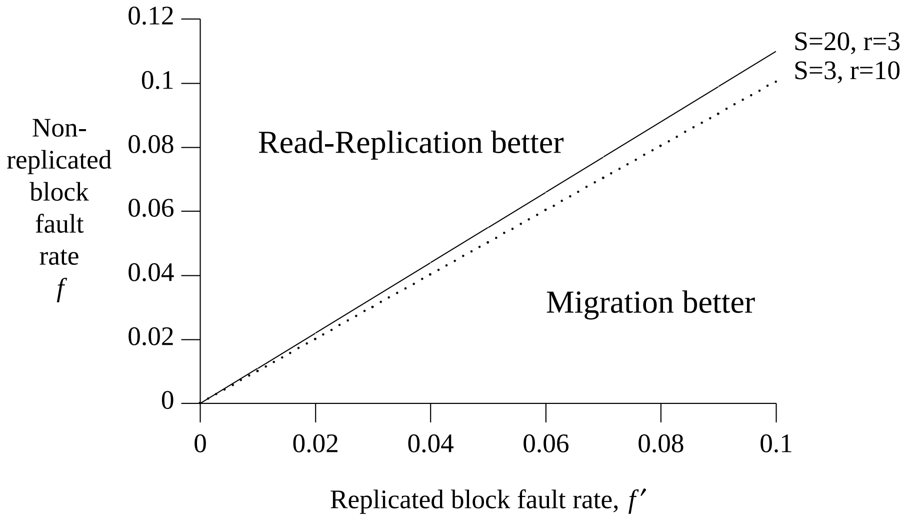


Figure 6: Performance comparison: migration vs. read replication

$$f' = \frac{4(1 - \frac{1}{S})}{44 + \frac{S}{r+1}}$$

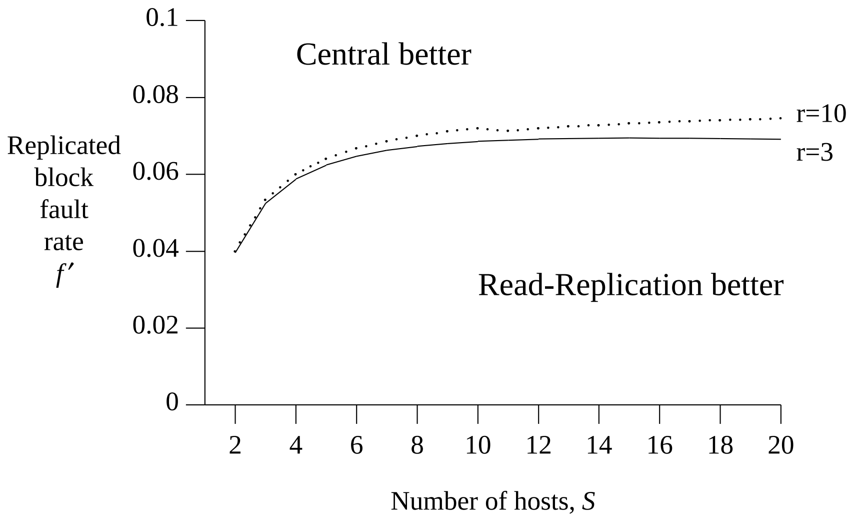


Figure 7: Performance comparison: central server vs. read replication

$$f' = \frac{S+2}{S+44(r+1)}$$

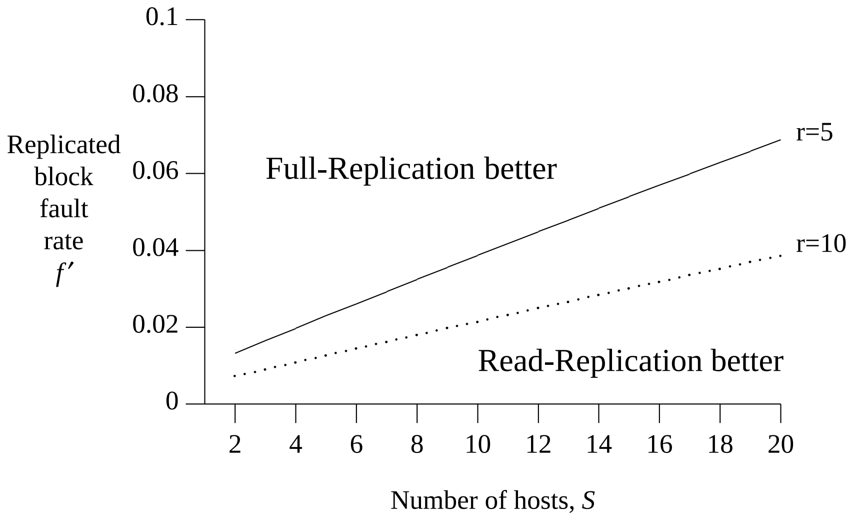


Figure 8: Performance comparison: read replication vs. full replication

of the two algorithms depends on a number of factors, including the degree of replication, the read/write ratio, and the degree of locality achievable in read replication. The full-replication algorithm is not susceptible to poor locality, since all data is replicated at all sites. On the other hand, the cost of multicast increases with S. Therefore, full replication performs poorly for large systems and when update frequency is high (i.e., when r is low).

Central server vs. full replication

These two algorithms represent the two extremes in supporting shared data: one is completely centralized, the other is completely distributed and replicated. Except for small values of S, the curve shown in Figure 9 is almost linear. For S values of up to about 20, the aggressive replication of full replication seems to be advantageous, as long as the read/write ratio is five or higher. For very large replication, however, the update costs of full replication catches up, and the preference turns to the simple central-server algorithm.

Remaining comparisons

The two remaining pairs not yet considered are summarized as follows. The comparison between central server and migration is very similar to that between central server and read replication, with a rather flat curve beneath $f = 0.09$. Thus, unless the block fault rate is very high, migration performs better. The comparison between migration and full replication reveals no clear winner, as in the case of read replication vs. full replication, with the key deciding factors being S and r.

$$r = \frac{1}{4}[(S-1) + \frac{3}{S-1}]$$

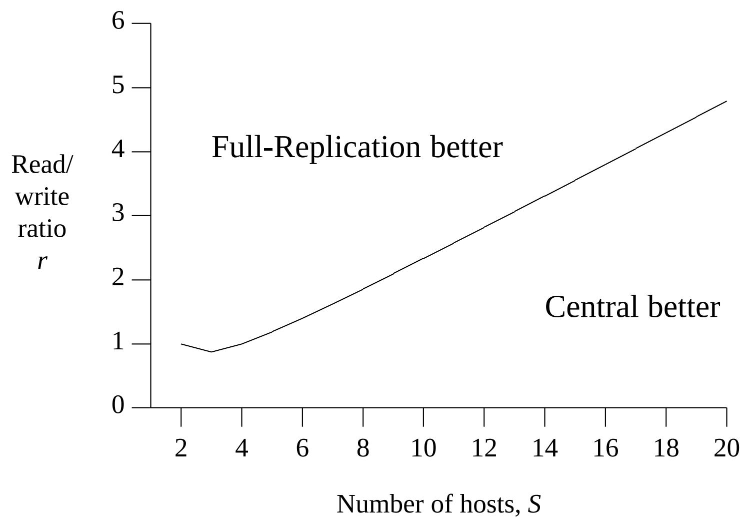


Figure 9: Performance comparison: central server vs. full replication

Comments

Based on the comparisons above, we can make a few observations. The central-server algorithm is simple to implement and may be sufficient for infrequent accesses to shared data, especially if the read/write ratio is low (i.e., a high percentage of accesses are writes). This is often the case with locks, as will be discussed in a later section. However, locality of reference and high block hit ratio are present in a wide range of applications, making block migration and replication advantageous.

The fault rate of the simple migration algorithm may increase due to interleaved accesses if different data items that happen to be in the same block are accessed by different sites. It thus does not explore locality to its full extent. The full-replication algorithm is suitable for small-scale replication and infrequent updates.

in contrast, the read-replication algorithm is often a good compromise for many applications. The central-server and full-replication algorithms share the property of being insensitive to access locality, so they may out-perform the read-replication algorithm if the application exhibits poor access locality.

A potentially serious performance problem with algorithms that move large data blocks is *block thrashing*. For migration, it takes the form of moving data back and forth in quick succession when interleaved data accesses are made by two or more sites. For read replication, it takes the form of blocks with read-only permissions being repeatedly invalidated soon after they are replicated.

Such situations indicate poor (site) locality in references. For many applications, shared data may be allocated and the computation may be partitioned to minimize thrashing. Application controlled locks may also be used to suppress thrashing (see Section 4.3). In either case, the complete transparency of the distributed shared memory is compromised somewhat.

3.3 Applications

From the above comparisons, it is clear that there are strong interactions between the shared data access patterns of applications and their expected performance using the various algorithms. To make our discussion more concrete, we study such interactions further in this section based on our experience implementing the algorithms and measuring their performance. We do this by examining a few types of applications and the consistency algorithms that might be suitable for them.

Numerical subroutines and other applications

The Blas-III package contains a frequently-used set of matrix manipulation subroutines implemented in Fortran. Typically, one or more (large) matrices are used as input to generate a result matrix of the same dimension. The amount of computation is usually substantial, and can be sped up by assigning processors to compute subregions of the result matrix. This results in the input data being widely read-shared, and the regions of the result matrix being updated by a one site.

Thus, read replication is highly desirable, whereas update multicast for write replication is unnecessary and too costly. It is apparent that the read-replication algorithm is the best. We observed excellent speedup using this algorithm. Li has studied similar applications in his thesis, and also reported very good speedup [11].

Interestingly, this data access pattern is wide spread. For example, we converted a graphics application to use distributed shared memory. This application uses a 3D description of a set of objects to compute the visible scene. Again, the scene description is input to the master thread, and read-shared among all the slave threads, which compute parts of the scene. Once the parts are completed, the master displays the entire scene. Like matrix computation, there is very little interference between the slaves except at the block boundaries of the scene buffer.

A parallel version of a printed circuit board inspection program exhibits very similar data access pattern, as well as excellent speedup using the read-replication algorithm.

Shortest paths

Similar to the above applications, a large matrix is used to represent the paths between pairs of nodes in a graph. However, elements of this matrix are updated *in-place* as new, better paths are discovered. In our straightforward parallel implementation, the threads make interleaved and mixed read and write accesses to the matrix, with the update rate being high at the beginning, and declining as better paths become harder and harder to find. The central-server algorithm is inappropriate because, like the applications above, accesses are very frequent.

On the other hand, since access locality is poor, large numbers of block transfers due to thrashing are unavoidable if either the migration or the read-replication algorithm is used. Full replication appears to perform better, especially for the later stages of the computation. Instead of transferring or invalidating a whole block when one entry is updated, only that entry is distributed.

Hypermedia and distributed game playing

Although these two types of applications serve very different purposes, they often share the characteristics of making interactive, concurrent accesses to shared data, with updates to moving, focused regions of the data space as the participants cooperate on or fight over the same areas. The read-replication algorithm may exhibit block thrashing as a result, and full replication again shows its merits.

Distributed locks

Locks are often used in parallel applications for synchronization. Typically, locks require little storage and exhibit poor locality, so that algorithms using block transfers — migration and read Replication — are inappropriate. If the demand on a lock is light, a thread will usually find the lock free and may simply lock it, perform the relevant operation, and then unlock it.

Since such operations are relatively infrequent, a simple algorithm such as central server is sufficient. However, if a lock is highly contended for, a thread may repeatedly attempt to lock it without success, or a "call-back" mechanism may be used to avoid spinning. In either case, the cost of accessing remotely stored locks can be significant, and migrating this lock alone is desirable. Some specialized algorithm seems to be desirable for distributed locks.

4 Improving performance

Many variations to the basic distributed shared memory algorithms exist. Many of them improve performance for particular applications by optimizing for particular memory access behaviors, typically by attempting to reduce the amount of communication, since costs are dominated by communication costs. Here, we describe two particularly interesting variations and also describe how applications can help improve performance by controlling the actions of the distributed shared memory algorithm. The largest improvements to performance can probably be achieved by relaxing consistency constraints [4], something we do not consider here.

4.1 Full replication with delayed broadcasts

The full-replication algorithm incurs $S + 2$ packet events on each write operation, where S is the number of participating sites. A slight modification to this algorithm can reduce the number of packet events per write to 4, while read operations continue to be executed locally (with no communication overhead).

Instead of broadcasting the data modifications to each site, the sequencer only logs them. A write sent to the sequencer by process A is acknowledged directly and a copy of all modifications that arrived at the sequencer since the previous time A sent a write request is included with the acknowledgement. Thus, the shared memory at a site is updated only when a write occurs at that site. As in the full-replication algorithm, read operations are performed locally without delays.

This variation on the full-replication algorithm still maintains consistency, but has at least two disadvantages, however. First, it refrains from updating shared memory for as long as possible and therefore does not conform to any real time behavior. Programs cannot simply busy-wait, waiting for a variable to be set. Second, it places an additional load on the (central) sequencer, which must maintain a log of data modifications and eventually copy each such modification into a message $S - 1$ times.

4.2 An optimistic full-replication algorithm

All of the basic algorithms described in Section 2 are *pessimistic* in that they ensure *a priori* that a process can access data only when the shared data space is and will remain consistent. Here, we describe an algorithm [9] that is *optimistic* in that it determines *a posteriori* whether a process has accessed inconsistent data, in which case that process is *rolled back* to a previous consistent state.

This algorithm evolved from attempts to increase performance of the full-replication algorithm by coalescing multiple write operations into a single communication packet.

Instead of obtaining a sequence number for each individual write operation, a sequence number is obtained for a series of consecutive writes by a single process. Obviously, this may lead to inconsistent copies of data, since writes are made to the local copy and only later transmitted (in batch) to other sites. If the modified data is not accessed before it reaches a remote site, then temporary inconsistencies will not matter. Otherwise a conflict has occurred, in which case one of the processes will have to roll back.

To roll back a process, *all* accesses to shared data are logged. To keep the size of these logs manageable (and the operations on them efficient), it is convenient to organize the data accesses into *transactions*, where each transaction consists of any number of of `read` and `write` operations bracketed by a `begin_transaction` and `end_transaction`.

When `end-transaction` is executed, a unique gap-free *sequence number* for the transaction is requested from a central sequencer. This sequence number determines the order in which transactions are to be committed. A transaction, **A**, with sequence number **n** is aborted if any concurrently executed transaction with a sequence number smaller than **n** has modified any of the data that transaction **A** accessed. Otherwise, the transaction is committed and its modifications to shared data are transmitted to all other sites; these transactions will never have to roll back. The logs of a transaction can then be discarded.

Clearly the performance of this optimistic algorithm will depend on the access behavior of the application program and the application program's use of transactions. If rollbacks are infrequent, it will easily out-perform the basic full-replication algorithm. It will also out-perform the read-replication algorithm for those applications that suffer from thrashing, since the optimistic algorithm compares shared memory accesses at the per data item level (as opposed to a per data block level). The primary drawback of this algorithm, however, is the fact that the shared memory is no longer transparent to the application, because the memory accesses must be organized into transaction and because roll backs must be properly handled by the application.

4.3 Application-level control with locking

Locks can be used by the applications not only for its synchronization needs, but also to improve the performance of the shared memory algorithms. For example, in the case of the migration and read-replication algorithms, locking data to prevent other sites from accessing that data for a short period of time can reduce thrashing.

In the case of the full-replication algorithm, the communication overhead can be reduced if replica sites only communicate after multiple writes instead of after each write. If a write lock is associated with the data, then once a process has acquired the lock, it is guaranteed that no other process will access that data, allowing it to make multiple modifications to the data and transmitting all modifications made during the time the lock was held in a single message without causing a consistency problem. That is, communication costs are only incurred when data is being unlocked rather than every time a process writes to shared data.

Having the application use locks to improve the performance of the shared memory has a number of disadvantages. First, the use of locks needs to be directed towards a particular shared memory algorithm; the shared memory abstraction can no longer be transparent. Second, the application must be aware of the shared data it is accessing and its shared data access patterns. Finally, in the case of those algorithms that migrate data blocks, the application must be aware of the block sizes and the layout of its data in the memory.

5 Concluding Remarks

Despite the simplifying assumptions made in the performance analyses, the essential characteristics of the algorithms are captured in the models used. The concept of distributed shared memory is appealing, because, for many distributed applications, the shared memory paradigm leads to simpler (application) programs than when data is passed directly using communication primitives. Moreover, with respect to performance, numerous implementations have shown that distributed shared memory can be competitive and, in some cases, even out-perform data passing programs.

On the negative side, the performance of the algorithms that implement distributed shared memory are sensitive to the shared memory access behavior of the applications. Hence, as we have shown, no single algorithm for distributed shared memory will be suitable for most applications. The performance-conscious application writer will need to choose an appropriate algorithm for his application after careful analysis or experimentation.

In some cases, he will want to use different algorithms (for different data) within a single application. Moreover, because these algorithms are sensitive to the access behavior of the applications, it is possible to improve their performance significantly either by fine-tuning the application's use of the memory or by fine-tuning the shared memory algorithm for the access behavior of the particular application, eliminating the advantages of transparent shared memory access.. It should also be emphasised that distributed shared memory may be entirely unsuitable for some applications.

Further work is still necessary to make distributed shared memory as versatile as their data passing counterparts. For example, the distributed shared memory algorithms we have described are not tolerant of faults. Whenever a host containing the only copy of some data items crashes, critical state is lost. Although the Central Server and the Full-Replication algorithm can be made tolerant of single host crashes (for example, by using a backup server in the case of the Central Server algorithm), it is not clear how to make the Migration and Read-Replication algorithms equally fault tolerant.

Compared to data passing, distributed shared memory does not appear to be as suitable for heterogeneous environments at this time, although several research efforts on this problem are currently underway [7, 13]. Consider, for example, the Migration algorithm in an environment consisting of hosts that use different byte orderings and floating point representations.

When a page is migrated between two hosts of different types, the contents of the page must be converted before it can be accessed by the application. It is not possible for the distributed memory system to convert the page without knowing the type of the application-level data contained in the page and the actual page layout. This complicates the interface between the memory system and the application.

If non-compatible compilers are used for an application to generate code for the different hosts such that size of the application-level data structures differ from host to host, then conversions on a per page basis become impossible. An additional problem for, say, numerical applications is that since the application has no control over how often a block is migrated or converted and accuracy is lost on every floating point conversion, the result may become numerically questionable.

For these reasons, we consider distributed shared memory to be a useful paradigm for implementing a large class of distributed applications, but do not expect it to become widely available in the form of a single standardized package, as has been the case for remote procedure calls, for example. Rather, we expect that distributed shared memory will be made available in a number of forms from which the application writer can choose.

Acknowledgments

Many thanks go to Tim McInerney, who did most of the work implementing the Migration and Read-Replication Algorithms for SMI Sun and DEC Firefly workstations, and to Orran Krieger who designed and implemented the Optimistic Full-Replication algorithm. The anonymous reviewers and the editors provided numerous valuable suggestions for improvements.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 37(8):921–929, August 1988.

[2] J. Archibald and J.L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. on Computing Systems*, 4:273–298, Nov 1986.

[3] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[4] D.R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems. In *Proc. 6th Int'l Conf. Distributed Computing Systems*, pages 190–197, May 1986.

[5] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–22, February 1988.

[6] B.D. Fleisch and G.J. Popek. Mirage: A coherent distributed shared memory design. In *Proc. 12th ACM Symp. Operating System Principles*, pages 211–222, 1989.

[7] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for Mach. In *Proc. 1989 Winter USENIX Conf.*, pages 229–243, January 1989.

[8] R.E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proc. 9th Intl. Conf' on Dist. CompSys.*, pages 498–505, June 1989.

[9] O. Krieger and M. Stumm. An optimistic apoproach for consistent replicated data for mulitcomputers. In *Proc. 1990 Hawaii Int;l Conf on System Sciences*, pages vol. 2: 367–375, 1990.

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers.*, pages 690–691, 1979.

[11] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University Dept. of Computer Science, 1986.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Comp. Systems*, 7(4), November 1989.

[13] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environment. In *Proc. 10th Int'l Conf. Dist. Computing Systems*, 1990.