SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(1 1), 985–1010 (NOVEMBER 1992)

A Comparison of Two Paradigms for Distributed Shared Memory

WILLEM G. LEVELT, M. FRANS KAASHOEK, HENRI E. BAL
AND ANDREW S. TANENBAUM
Department of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

SUMMARY

Two paradigms for distributed shared memory on loosely-coupled computing systems are compared: the shared data-object model as used in Orca, a programming language specially designed for loosely-coupled computing systems, and the shared virtual memory model. For both paradigms two systems are described, one using only point-to-point messages, the other using broadcasting as well.

The two paradigms and their implementations are described briefly. Their performances are compared on four applications: the travelling-salesman problem, alpha-beta search, matrix multiplication and the all-pairs shortest-paths problem. Measurements were obtained on a system consisting of 10 MC68020 processors connected by an Ethernet. For comparison purposes, the applications have also been run on a system with physical shared memory. In addition, the paper gives measurements for the first two applications above when remote procedure call is used as the communication mechanism.

The measurements show that both paradigms can be used efficiently for programming large-grain parallel applications, with significant speed-ups. The structured shared data-object model achieves the highest speed-ups and is easiest to program and to debug.

KEY WORDS Amoeba Distributed shared memory Distributed programming Orca Shared data-objects Shared virtual memory

INTRODUCTION

As computers become cheaper, there is an increasing interest in using multiple CPUS to speed up individual applications. There are basically two design approaches to achieve this goal of high performance at low cost: multiprocessors and multicomputers. Multiprocessors contain physical shared memory; processors in a multiprocessor can communicate by reading and writing words in this memory. Multicomputers, on the other hand, do not contain physical shared memory; processors communicate by exchanging messages. In this paper, we compare two hybrid approaches that allow processors in a multicomputer to communicate through *distributed shared memory.* We shall discuss the implementation of these two hybrid approaches, describe some applications we have written on both, and present measurements of the performance.

The key property of a multiprocessor is that each processor has a consistent view of the physical shared memory. When any processor writes a value to memory, any other processor that subsequently reads the word will retrieve the value just written. To achieve these semantics special and complicated hardware (such as snoopy caches

0038–0644/92/110985–26$18.00

Received 14 August 1990
Revised 22 March 1991 and 21 July 1992

or fast switching networks) is needed to build a multiprocessor. The software for multiprocessors, however, is straightforward. All processes run within a single address space, so they can easily share data structures and variables. When one process updates a variable, other processes will immediately see this change. The methods to allow processes to synchronize their activities are well understood. Multiprocessors are hard to build but are relatively easy to program.

In a multicomputer each processor has its own private memory which it alone can write to and read from. These processors can be connected using standard networking technology. The software for multicomputers, on the other hand, is much more complicated. Processors in multicomputers must communicate using message-passing. Although much effort is put into hiding the message-passing from the programmer it is difficult to make it entirely transparent. A very popular approach to hide the communication is the remote procedure call (RPC). The idea is to make the communication look like an ordinary procedure call. The programmer, however, still has to be aware that the semantics of remote procedure calls are different from those of local procedure calls. For example, passing pointers as parameters in an RPC is difficult, and passing arrays is costly. This makes programming on a multicomputer difficult.

To combine the advantages of multiprocessors (easy to program) and multicomputers (easy to build), communication paradigms that simulate shared data on a multicomputer have become popular. These mechanisms are implemented with message-passing, but they provide the illusion of shared data. They provide a *distributed shared memory* (DSM). Processes on different processors run in the same address space. All processes can access the variables in the shared address space directly. They communicate through these variables.

In this paper we compare two approaches to distributed shared memory: the shared virtual memory model (SVM) [1,2] and the shared data-object model. [3] Both models provide logical shared memory, but use different abstraction techniques. Shared virtual memory provides the illusion of true physical shared memory. The shared data-object model encapsulates shared data in user-defined objects.

Shared virtual memory

The shared virtual memory model simulates true physical shared memory on a loosely-coupled system. A number of processes share a single address space. This address space is divided into pages, which are distributed among the processes. Processes either have *no*, *read* or *write* access to a page. *Read* pages can be replicated on multiple processors to reduce access times. The system provides a coherent address space: a read operation always returns the value of the most recent write to the same address. Mutual-exclusion synchronization can be implemented by locking pages.

The SVM paradigm can be viewed as a low-level *unstructured* DSM approach. The address space is divided into fixed-size pages with no relation to the structure of the stored data. The SVM system is completely transparent to the processes that use it. There is no distinction between shared and non-shared data. Even the operating system itself can use it for communication between processors.

A disadvantage of this low-level approach is the lack of support for the application programmer. The SVM can only be accessed with primitive operations, such as load,

store and lock. When two independent and heavily used variables reside on the same page, this will lead to contention. To avoid unacceptable performance penalties, such variables have to be placed on different pages. This has to be done by the programmer, because a compiler has not enough knowledge to decide this. The compiler does not know how variables are mapped onto pages and it cannot decide at compile time if two objects that are possibly accessed through pointers are independent. For popular languages, such as C, this problem becomes even harder for memory that is allocated during run-time. The compiler has no way of knowing how this memory is going to be used and whether two blocks of dynamically allocated memory will be used independently of each other. Therefore the application programmer must be constantly aware of how the data-structures are accessed and where they are placed in the SVM address space, or suffer an unacceptable performance penalty. Thus in practice the shared virtual memory is not really transparent to the user, unless the user does not care about performance. Furthermore the SVM is just one global flat address space; no access protection or type-security is enforced by the system. This makes distributed programming difficult.

Shared data-object model

The shared data-object model is a high-level, *structured,* approach to distributed shared memory. In contrast to the SVM model, which is implemented by the kernel using hardware support, the shared data-object model is implemented outside the kernel completely in software. A run-time system (RTS), using information generated by the compiler, keeps the DSM coherent.

In a shared data-object language, shared data are encapsulated in objects. A shared data-object is an instance of a user-defined abstract data type and can only be accessed through operations defined in the object's specification. These operations are executed indivisibly and the RTS ensures that all processes that share the object see the result. Partitioning of the DSM address space is not defined by the system, as in the SVM approach, but implicitly by the application programmer. The unit of programmer-defined sharing is the shared object, not the page. As an object is an instance of an abstract data type, variables that are independent of each other will typically reside in different objects. False sharing as in SVM will be less of a problem in the shared data-object model.

One way to share objects in a shared data-object language is by dynamic creation of processes. The parent process can pass any of its objects as a shared object to its children, which communicate through these shared data-objects. The RTS may replicate these objects on more than one processor, to reduce the access time. Other sharing mechanisms are also possible.

Also in contrast to the SVM approach, the distributed shared memory is not treated as a flat address space that can be accessed in an arbitrary way. The semantics of the language restrict the scope of shared variables, similar to scope rules in a sequential language. The processes can only access those shared data-objects they are supposed to access. Furthermore, the shared data can only be accessed with high-level operations, which the programmer defines in the abstract data type. Because the execution of these operations is indivisible, mutual exclusive access is provided implicitly. Table I summarizes the differences between these two paradigms.

Table I. Differences between the SVM and shared data-object paradigms

	Shared virtual memory	Shared data-object model
Implementation level	In kernel, using hardware support	Completely in software
Unit of sharing	System defined page	User defined object
Unit of synchronization	Machine instruction	Procedure
Data placement	Explicit	Implicit
Address space	Flat	Structured

Related work

Several systems that provide shared data on distributed computing systems have been designed. Most of them do not provide a structured address space, but just a flat one. Examples of these are page-based systems such as IVY, [4] Shiva, [5] Mirage, [6] Mether [7,8] and Munin. [9] Each one has its own coherency technique, but they are all based on the ideas Li put forward in his thesis. [1] For example, Mirage uses a technique similar to Li's fixed distributed management algorithm, but as an extra feature it maintains timers to prohibit pages to be paged out too soon after paging in. Other unstructured distributed shared memory systems include the Agora shared memory, [10] where the shared data consists of immutable data elements, which can be accessed through mutable maps. Data are changed by adding new data elements and changing the map. In Agora, however, read operations can return stale data.

An example of a structured DSM system is the tuple space used in Linda. [11,12] The tuple space contains tuples, which are similar to records. A tuple can only be changed by taking it out of the tuple space, modifying it locally, and reinserting it. Another example of a structured DSM system is Sloop, which supports a shared object space. [13] A discussion of these and other distributed shared memory paradigms is provided by Bal and Tanenbaum. [14]

Outline for this paper

The purpose of this paper is to compare two paradigms for distributed shared memory: a structured and an unstructured one. Does it pay to implement an application-oriented DSM system, such as the shared data-object model, or is the general approach of the shared virtual memory equally good? To make a direct comparison possible, we have implemented these two DSM paradigms on the same distributed computing system, each in two different ways. Using these systems, we have measured the performance of four applications: the travelling-salesman problem (TSP), alpha–beta search, matrix multiplication and the all-pairs shortest paths problem. To place these measurements in perspective, we also give the measurements obtained by running these applications on a physical shared memory system and we supply measurements for TSP and alpha–beta using pure message-passing as the communication mechanism. As a side-effect of this research we can present real-time performance figures for two SVM implementations. We shall, furthermore, discuss some programmability aspects of these two paradigms.

The remainder of this paper is structured as follows: the next section describes the hardware and the operating system (Amoeba) we have used for our experiments.

Then we discuss the two SVM systems we have implemented. The following section describes the shared data-object model implementions. Then we describe the applications we have implemented, after which we discuss the measurements and other aspects of the two paradigms. In the last section we present our conclusions.

THE AMOEBA DISTRIBUTED OPERATING SYSTEM

Amoeba is an operating system specially designed for loosely-coupled computing systems. [15] The Amoeba architecture consists of four principal components.

First are the workstations, one per user, on which users can carry out editing and other tasks that require fast interactive response. We are currently using Sun-3s as workstations. Second are the pool processors, a group of CPUs that can be allocated dynamically as needed, used, and then returned to the pool.

Third are the specialized servers, such as directory servers, file servers, database servers, bank servers, boot servers, and various other servers with specialized functions.

Fourth are the gateways, which are used to link Amoeba systems at different sites and different countries into a single, uniform system. The main function of the gateways is to isolate users from the peculiarities of the protocols that must be used over the wide-area networks.

Amoeba is an object-based system. The system can be viewed as a collection of objects, on each of which there is a set of operations that can be performed. The list of allowed operations is defined by the person who designs the object and who writes the code to implement it. Both hardware and software objects exist.

Associated with each object is a *capability*, a kind of ticket or key that allows the holder of the capability to perform some (not necessarily all) operations on that object. Capabilities are protected cryptographically to prevent users from tampering with them.

The Amoeba kernel runs on each processor. It provides communication service and little else. Communication between processes is done with remote procedure calls (RPC). An RPC consists of the following steps: a client process sends a request to a server process, and blocks. The server accepts the request, processes it, and sends a reply message back. At the arrival of this reply an acknowledgement packet is sent, the client is unblocked again and the RPC has succeeded. A message consists of a header of 32 bytes and a buffer of up to 30,000 bytes.

In addition to RPC, the Amoeba kernel provides reliable, order-preserving, broadcast communication. [16,17] Even in the presence of communication failures, the kernel guarantees that if two processes broadcast a message simultaneously, one of them goes first and its message will be processed at all processors before the other one. In normal operation, this protocol needs two messages per broadcast.

A process consists of a collection of threads that run on the same processor. These threads share a single address space, but they all use a dedicated portion of this address space for their private stack. In this version of Amoeba (3.0) threads are not pre-empted; conceptually they run until they block.

UNSTRUCTURED DSM: THE SHARED VIRTUAL MEMORY PARADIGM

The shared virtual memory model simulates true physical shared memory on loosely-coupled systems. This simulation is accomplished by dividing the SVM address space into fixed-size pages, which are distributed among the processes forming an application. The SVM system co-ordinates the movement and validity of the pages. Pages with read-only access can be replicated at multiple processors to reduce access times. An invalidation scheme is used to keep the address space coherent. For example, when a process tries to write on a page, all other copies of that page are first invalidated, then the write is permitted.

To synchronize multiple requests for the same page, each page is owned by a specific process (which may change in time). This owner process has a valid copy of the page, and all requests for that page are sent to it. To be able to keep the address space coherent, the SVM system must, for each page, keep track of the owner process and of all the processes containing a valid copy.

Li has proposed several methods for organizing and maintaining this ownership and copy-list data.[1] He first made a distinction between centralized and distributed management of ownership data:

1. With centralized management, one process, the central manager, keeps track of all pages. All requests for a page are first sent to the central manager, which in turn sends it to the current owner of the page. In the regular *central manager* method, the central manager keeps track of the copy-list data too. In the *improved central manager* method, the copy-list information is decentralized; it is stored at the process that owns the page.
2. With distributed management, the ownership data is distributed among the processes. This can be done in a fixed or dynamic way. *Fixed distributed management* means that each process manages the ownership data of a predetermined subset of the pages, which remains fixed throughout the whole lifetime of an application. Copy-list data can either be stored at the manager or at the owner. In the *dynamic distributed manager* approach, ownership (and copy-list) data is transferred together with the page itself. Only the owner process knows for sure who the owner is.

An important design choice is the page size. The page size is a trade-off between the cost of sending a message and memory contention. Because of set-up costs, the time to send a message is not proportional to the size of the message. This favours large page sizes. On the other hand, the bigger the page, the greater the chance that two or more key variables used by different processors will accidentally reside there, leading to contention.

Which page size is optimal is an open question; it will probably vary from system to system and from application to application. IVY uses lK, Shiva uses 4K, Mirage uses 512 bytes and in Mether the application can choose between two page sizes: 32 bytes or 8K. The Amoeba operating system internally uses a page size of 4K for the partitioning of memory space. We therefore decided to use 4K as the page size too.

We have implemented two SVM systems: the first using centralized management, the second using dynamic distributed management. The first system uses RPC for interprocess communication. The second uses broadcasting as well as RPC. We now describe each system in turn.

Central manager system

In our implementation, every application runs in its own private SVM address space, managed by its own central manager. The system consists of two parts: one central manager process and a collection of SVM handler threads, one associated with each process in the application. The central manager is an ordinary user program that handles the bulk of the protocol. All requests for pages are sent to the central manager, which keeps track of page-ownership and copy-list data. Each SVM handler manages the page-access rights for its associated process in the application. The SVM handler threads run in kernel space, because they have to access kernel data structures (e.g. the page table). This is shown in Figure 1.

The central manager maintains a table with copies of pages to which no process has write access. Requests for such a page can be handled without contracting the owner. The central manager can reply immediately with the copy it stored earlier.

On receiving a write request, the central manager has to invalidate all other copies of that page. If this were handled by a single thread, the copies would be invalidated one by one, which is slow. Therefore, the central manager has a thread for each SVM handler in the application. These threads are used to send the invalidation messages in (pseudo) parallel. In this way, the next invalidation RPC can be started before the previous one has been completed.

When the central manager receives a write-page request from a process that still has a valid read-copy, the central manager does not send a copy of the page. Thus a read fault, followed immediately by a write fault on the same processor, causes only one page transfer.

A typical communication pattern caused by a write fault in one process, followed by a read fault in a different process (on the same page), is shown in Figure 2. At first (top of the figure), processes 2 and 3 have a read-copy of the page. Process 1

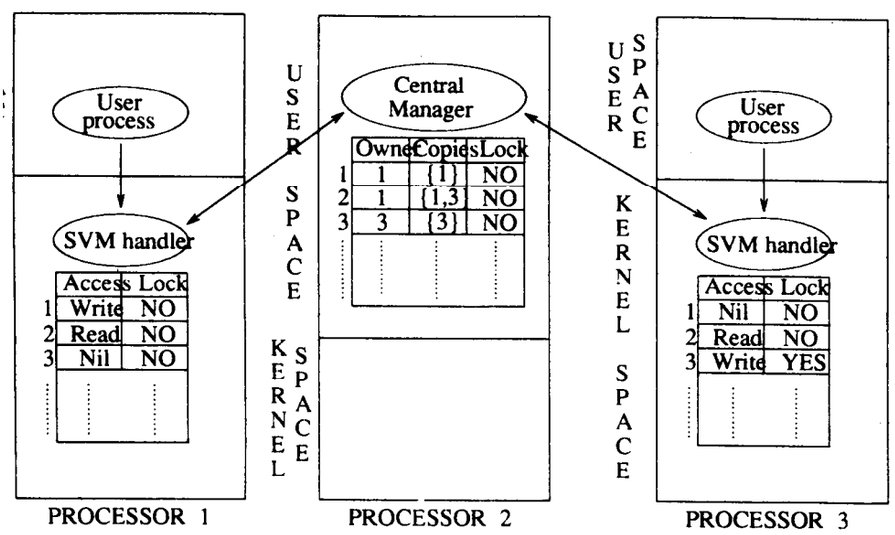

Figure 1. Processor outline in the central manager system

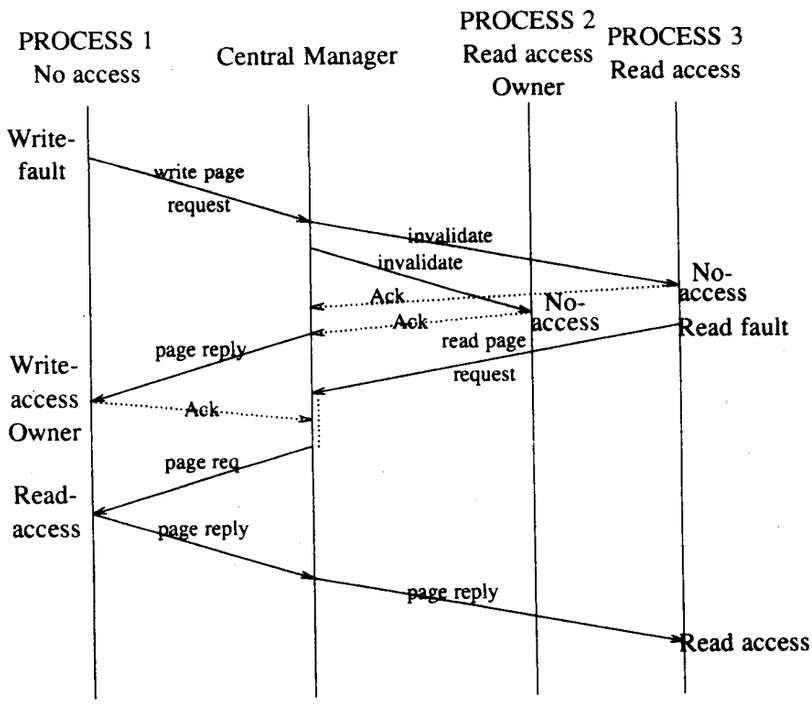

Figure 2. A write fault in process 1, followed by a read fault in process 3 *in the central manager system*

tries to write to the page, which causes a write fault. The SVM handler of process 1 sends a write request to the central manager. The central manger has a valid copy available, so it only has to invalidate the other copies. When both copies are invalidated, it can send the page to process 1.

Shortly after the read-copy at process 3 is invalidated, a read fault occurs, and a read-page request is sent to the central manager. Because the central manager is still handling the request from process 1, stores it locally, and sends it to acknowledgement from process 1 arrives, the central manager can handle the blocked request. It requests a read-copy from process 1, stores it locally, and sends it to process 3.

For mutual exclusion synchronization a page-locking mechanism is provided. A process can lock a page for reading or for writing. A read-locked page will not be invalidated until it is unlocked again,. Write locking is more powerful: it provides a process with the sole copy of a page. Upgrading of a read-lock to a write-lock is not possible.

When a page is not locked, a page fault takes two RPCS to complete: the first from the requester to the central manager, the second from the central manager to the owner. When the central manager has a valid copy, just one RPC is needed. In addition, on a write fault, an extra RPC is necessary for each copy of the page.

There are two important differences between this central manager system and the system proposed by Li. First, in Li's central manager system, one of the processes

in the application also works as the central manager. In our system, the central manager is a distinct process, which does not work on the application.

Our approach has the advantage that the central manager can run entirely in user-space. The kernel is therefore not enlarged by this code. This fits the Amoeba principle of keeping the kernel small. However, implementing the central manager as a 'distinct process in user-space has a disadvantage too, which is caused by the non-preemptive scheduling of processes in Amoeba 3.0.

In Amoeba, user-processes are rescheduled twice a second, or when they execute a system call. When the central manager has to service a request, it is unacceptable that it has to wait for rescheduling. Therefore, the central manager must be the only process running on a processor, so no other user-processes can delay it.

The second difference is that Amoeba uses remote procedure calls as its communication mechanism, whereas Li uses straight message-passing. The communication pattern in Li's (improved) central manager system, in case of a page fault, is as follows:

1. A message from the faulting process to the central manager.
2. A message from the central manager to the owner process.
3. A message from the owner to the faulting process.

This is not appropriate for RPC, where a process always waits for a reply. The faulting process contacts the central manager with an RPC, and it will block until the central manager sends a reply. The process cannot accept a reply from another process. The communication pattern therefore is as follows:

1. A request from the faulting process to the central manager.
2. A request from the central manager to the owner process.
3. A reply from the owner to the central manager.
4. A reply from the central manager to the faulting process.

These communication patterns are depicted in Figure 3. Here, the RPC uses four messages, whereas Li's scheme uses three. An Amoeba RPC uses three packets, a request, a reply and an acknowledgement. If the messages in Li's scheme use two packets (one for the message and one for the acknowledgement), both approaches use the same number of physical packets. When a copy of a page has to be transferred (needing a large message), Li's scheme only transmits the page once, whereas the

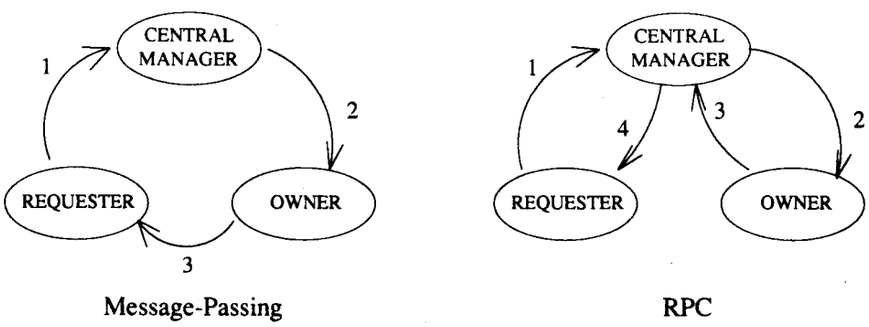

Figure 3. Communication patterns in IVY and our Amoeba implementation

RPC approach has to transmit it twice. Thus, our implementation of the central manager performs worse (due to the RPC) than Li's implementation in this respect.

Dynamic distributed management system

In our distributed management method each process has its own SVM handler, like the processes in the central manager method. But in contrast to the central manager method, there is no central manager. In the event of a page fault, the SVM handler associated with the faulting process broadcasts a request for the page. All other SVM handlers receive this broadcast. One of them is the owner and it will send the page to the requester using an RPC. From this point on, the requesting process is the new owner, and it will respond to the next request for this page. Thus, in contrast to the central manager method, where ownership only changes on a write fault, in the distributed management method ownership also changes on a read fault.

A write-request broadcast causes all SVM handlers to invalidate their copy of the page, so no copy-lists are needed. This scheme only works because each process gets the broadcasts in the same order, and no broadcasts are lost. All SVM handlers have the same view of the system and there can be no confusion over which process is the owner and which processes have a valid copy.

In this system, pages can be locked too. The SVM handlers try to handle a broadcast affecting a locked page as soon as possible. For instance: on receiving a write-request for a read-locked page, the SVM handler will send the page right away. The invalidation is done as soon as the page is unlocked again.

A page fault (read and write) takes one broadcast and one RPC to complete. When a read-copy is upgraded to a write-copy, no RPC is used. A typical communication pattern in case of a write fault is shown in Figure 4.

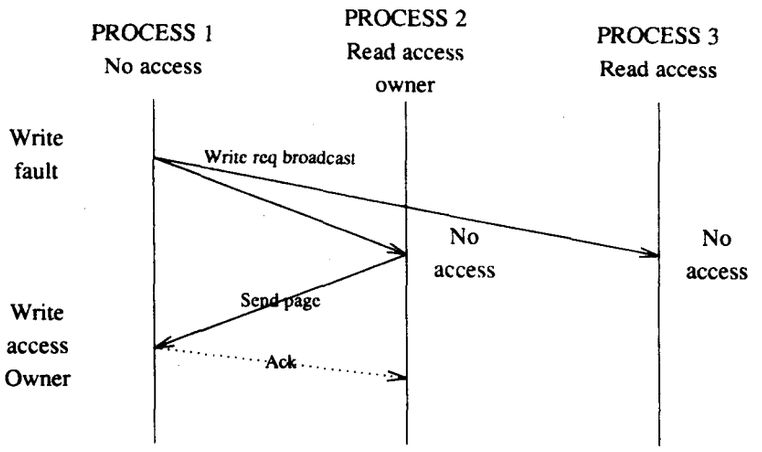

Figure 4. A write fault in the distributed management method

STRUCTURED DSM: THE SHARED DATA-OBJECT MODEL

The shared data-object model was proposed by Bal and Tanenbaum [14] to ease the programming of parallel applications. They and Kaashoek designed and implemented a new parallel programming language *0rca* [3,18–20] which uses the shared data-object model as communication paradigm.

In the shared data-object model, shared data are encapsulated in *data-objects,* * which are variables of user-defined abstract data types. An abstract data type has two parts:

(a) a specification of the operations that can be applied to objects of this type
(b) the implementation, consisting of declarations for the local variables of the object and code implementing the operations.

Instances (objects) of an abstract data type can be created dynamically. Each object contains the variables defined in the implementation part. These objects can be shared among multiple processes, typically running on different machines. Each process can apply operations to the object, according to their listing in the specification part of the abstract type. In this way, the object becomes a communication channel between the processes that share it.

The shared data-object model uses two important principles related to operations on objects:

1. All operations on a given object are executed *atomically* (i.e. *indivisibly).* To be precise, the model guarantees *serializability* [21] of operation invocations: if two operations are applied simultaneously to the same data-object, then the result is as if one of them is executed before the other; the order of invocation, however, is non-deterministic.
2. All operations apply to *single* objects, so an operation invocation can modify at most one object. Making *sequences* of operations on different objects indivisible is the responsibility of the programmer.

Linguistic support for the shared data-object model

The new programming language Orca gives linguistic support for the shared data-object model. Orca is a simple, procedural, type-secure language. It supports abstract data types, processes, a variety of data structures, modules and generics. It does not support global variables and pointers, but provides a new type-constructor graph that allows the programmer to build any data-structure that can be constructed with pointers. Parallelism in Orca is based on explicit creation of sequential processes. Processes are similar to procedures, except that procedure invocations are serial, but newly created processes run in parallel with their creator.

The parent can pass any of its *data-objects* as a shared parameter to its children. The parent and child can communicate through this shared object, by executing the operations defined by the object's type. The children can pass the objects to *their* children, and so on. This mechanism can be used for sharing objects among any number of processes. If any of these processes performs an operation on the object,

* We shall sometimes use the term 'object' as a shorthand notation for data-objects. Note, however, that unlike the situation in most parallel object-based systems, objects in this model are purely passive.

they all observe the same effect, as if the object were in shared memory, protected by a lock variable.

Processes in a parallel program sometimes have to synchronize their actions. This is expressed in Orca by allowing operations to *block* until a specified predicate evaluates to true. A process that invokes a blocking operation is suspended for as long as the operation blocks. The data-structuring mechanism in Orca is type-secure.

Implementations of the run-time system

A run-time system for Orca is, among other tasks, responsible for managing processes, shared data-objects, and (un)marshalling of data structures. The RTS works closely together with the compiler to perform its task. The compiler generates *descriptors* to allow the RTS to marshal complex objects, such as graphs, and to perform extensive run-time type checking. These descriptors describe the layout of an object and the sizes of the components of the object. For example, for a graph, such a descriptor contains a table with pointers to the actual nodes in the graph. The nodes in a graph are represented as offsets in this table, rather than as pointers, to make marshalling of a graph easy. Using the descriptors the RTS can marshal any object and check all array and graph references. If a node is deleted from a graph and subsequently referenced, a run-time error will be given.

Another important task that the compiler performs and that is used by the RTS, is the classification of operations. The compiler tells the RTS which operations change the internal data of an object (a write-operation) and which operations do not change the internal data (a read-operation). Using this information the RTS can implement efficient replication schemes. Bal *et al.* have experimented with two implementations of the RTS on a distributed computing system using different replication schemes.[22] We shall describe these two implementations briefly below.

Structure of the multicast RTS

The first implementation replicates all objects on all processors and uses a distributed update protocol based on reliable multicast messages for keeping all copies up to date. The RTS contains the same protocols as used above (for distributed management) for implementing reliable multicast on top of an unreliable network.

The multicast RTS is best thought of as a new kind of operating system kernel designed specifically for parallel applications. Shared data-objects are created and updated by the kernel. User processes can read local copies of objects directly, without using the kernel. If a user process applies a write operation to a shared object, the user process traps into the kernel; the local kernel multicasts the operation and its parameters to the kernels of all processors; each kernel then applies the operation to its local copy of the object. As the multicast primitive is reliable and indivisible, write operations are executed by all kernels in the same order.

Structure of the remote procedure call RTS

The second RTS is implemented on top of the Amoeba distributed operating system, It replicates and migrates objects selectively, based on statistical information gathered during run time. It updates the replicas through two-phase primary-copy

protocol, using point-to-point communication only. For each object, one processor contains the primary copy of the object and zero or more other processors contain secondary copies. Read operations are applied to the local copy, if available, and write operations are forwarded to the processor with the primary copy.

Updating replicas through point-to-point messages is more expensive than through multicast messages. Furthermore, the communication overhead with point-to-point messages grows linearly with the number of replicas. The RPC RTS therefore replicates objects *selectively*. A given processor only keeps a copy of an object if it reads the object frequently. Run-time statistics are maintained for deciding where to store the primary and secondary copies of each object. Application programmers are not aware of this; it is completely automatic.

There is one incarnation of the RTS on each processor. Each incarnation of the RTS starts a number of *listener tasks* that wait for requests from remote machines. A request can either be:

1. An update of a replicated object.
2. An operation to be performed on an object whose primary copy resides on this machine, on behalf of a remote machine.
3. A request to create a new Orca process.

When a message needs to be sent to another machine, the task wishing to send the message deposits it in a per-machine queue. For each queue—and thus for each remote machine—there is a *talker task* that handles the messages in the queue. A talker repeatedly waits for a message, sends it to the remote machine (using Amoeba RPC), and optionally returns a reply to the task that requested the message to be sent.

With this approach, the replicas of an object can be updated in parallel by depositing update messages in multiple queues. This programming model is similar to the *promises* model.[23] If each Orca process performed the RPC calls itself, parallel updating would not be possible, since RPC calls are blocking. As another advantage of our approach, multiple objects residing on the same machine can be updated in parallel.

EXPERIMENTS AND RESULTS

The most important properties of a distributed shared memory system are its performance and its ease of programming. In order to judge DSM systems these properties must be compared. The way of comparing programmability aspects is easy: just program a few applications. However, the interpretation is subjective. In contrast, performance comparison can be done objectively (using the system clock). But the question of what to compare is not straightforward.

Because the shared virtual memory system and our shared data-object system use different methods for replication and updating, it is not possible to compare low-level operations. We therefore decided to compare them by running four parallel applications: the travelling-salesman problem, alpha–beta search, matrix multiplication and the all-pairs shortest paths problem. To make a direct comparison possible, for all applications both systems used the same input, the same division of work and the same algorithm.

We do not include the time to start the processes. Time is measured from the

moment all processes have been started, to the moment that they all have finished their computations. The SVM programs are written in C, using two extra systems calls to lock and unlock the page(s) containing a shared variable.

(a) lock(variable, sizeof(variable), lock_type)
(b) unlock(variable, sizeof(variable), lock_type).

In the shared data-object model different shared variables are handled independently. In the shared virtual memory model, variables residing on the same page are always handled together. When the variables are independent, this can lead to unnecessary contention. To get good performance, it is necessary to place independent variables on different pages and variables that are mostly referenced together on the same page.

To emphasize the importance of good placement, we ran each test in the SVM system twice, once with all variables optimally distributed, and once with the shared variables packed together on as few pages as possible. We shall give performance measurements for both cases.

As described earlier, the central manager has to run on a separate processor. We counted this processor when calculating the speed-ups. The central manager measurements therefore start at two processors.

We ran all measurements of the SVM model at least four times. The variances of the measurements were all below 1·0 and mostly around 0·25. The measurements of the shared data-object model were run three times. For comparison purposes we also ran all applications on a multiprocessor with physical shared memory. In addition, we supply measurements for TSP and alpha–beta, programmed using remote procedure calls as the communication mechanism. How these were programmed is described by Bal *et al.* [24]

The Appendix lists all these timings. To keep the tables compact, we supply the mean values only. In addition to these tables there are a few graphs. They depict the speed-ups achieved by each of the six systems. For the SVM systems, we used the measurements achieved with good placement.

Hardware environment

All measurements were obtained on a distributed system consisting of up to 10 MC68020 CPUs, interconnected through Lance chips to a 10Mbit/s Ethernet. Each processor has 2 MB of private memory. For the physical shared memory system we used the same type of processors but now connected by a VME-bus to 8 MB of shared memory. Unfortunately, we only had nine working processors, so no measurements for 10 processors can be supplied. This system does not have snoopy caches, so every access to shared memory goes through the VME-bus.

In our Amoeba version, a 4-byte and a 4K RPC take 104 and 6·7 ms to complete, respectively. [15] A 20-byte broadcast to 10 processors, using hardware multicast, takes 1–5 ms to complete. [16]

Basic operations in the shared virtual memory system

In addition to taking measurements of applications, we also measured the timing of some basic operations in our two SVM systems. They are shown in Figure 5. The

Central Manager

Operation	Time (msec)
Get page, stored at manager	7.1
Get page, not stored at manager	14.0
Additional time to invalidate copies	
2 copies	4.4
4 copies	7.4
6 copies	10.4
8 copies	13.9

Distributed Management

Operation	Time (msec)
Get read or write page	10.2
Invalidate all copies of a page	3.0

Centralized & Distributed Management

Operation	Time (msec)
Lock & unlock an in-memory page	0.275
Page-fault schedule time to SVM-handler and back	0.267

Figure 5. Basic operations in the shared virtual memory systems

time to get a page is the time as seen by the user process on a page fault. Hence, it includes the time spent in the kernel and the time to enter and leave the kernel. These measurements were taken on an otherwise almost idle system.

Travelling-salesman problem

In the travelling-salesman problem (TSP), a set of cities and the distances between these cities are supplied. Given an initial city, the problem is to find the shortest route that visits every other city exactly once. This can be solved with a branch-and-bound algorithm in which a tree of possible solutions is built. Each node of the tree depicts a city, and a path in the tree describes a route through these cities that are depicted by the nodes on the path. A path from the root-node to a leaf-node describes a route that visits every city exactly once.

The bounding rule uses the length of the shortest route found so far. If the length of a path to a node exceeds this minimum length, all paths through this node will be longer than the shortest route, so they do not have to be searched.

The sequential branch-and-bound algorithm applies depth-first search on the tree, using the nearest-city-first heuristic. This can easily be parallelized. The search tree is static and subtrees can be searched independently by multiple processes. The only dynamic quantity is the length of the shortest route found so far. This variable (minimum) is not updated very often, perhaps 10 times during the whole search. It will be read very frequently, but that causes no overhead, because reading is a local operation.

We used as input an 12 × 12 distance matrix, and a starting city. The work is divided by a distribution process, which constructs the top two levels of the search tree, i.e. generating 11 × 10 = 110 partial paths. These partial paths are searched further (with sequential DFS) by worker processes (and the distribution process) until all paths have been inspected.

We used three randomly generated distance matrices. The measurements we supply are the mean values for these three graphs. Figure 6 depicts the speed-ups. Both systems achieve close to linear speed-ups. This is explained by the fact that, as soon as one of the workers discovers a new minimum, it updates the shared variable, and all workers can use this minimum to prune their subtrees earlier. Because in parallel search a low minimum is sometimes found earlier than in sequential search, it can happen that fewer nodes are searched. For an input matrix where a good minimum is found quickly, the speed-ups are in some cases superlinear. The RPC implementation performs worse, because each worker process can only accept a new value for the minimum when it has finished its previous job.

Alpha-beta search

Alpha-beta search is typically used for two-player, zero-sum, board games such as chess and draughts. It decides on the next move by evaluating the current board position.[25] To evaluate a board position, a tree is built with the current board position as the root-node and for every possible move, a child with the board position after that move. The child with the worst possible board position (which is a position for the opponent) indicates the best possible move. Before a value can be assigned

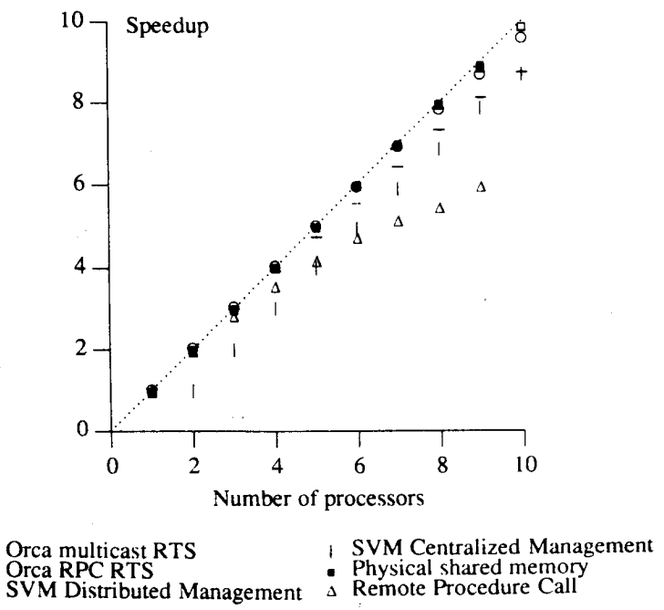

Figure 6. Travelling-salesman problem

to the root-node, all its children have to be evaluated. This is done by first evaluating their children, etc. This process must stop somewhere, so at a certain level (*ply*) a *static evaluation is* done on a position. This assigns a value (weight) to a position.

The whole procedure can be programmed compactly by using positive values at odd levels, and negative values at even levels. The value of a node is expressed as follows: assign to a node the value of the child with the lowest value, negated.

To reduce the search space, the search tree is pruned using an upper and lower bound, *alpha* and *beta.*

In the parallel implementation that we have used, the work is divided in a way similar to the TSP approach: each process searches a part of the search tree. In contrast to TSP, where reaching a leaf-node and finding a shortest route is enough to improve the solution, in alpha–beta search the value of a node can only be changed by combining the values of its children. To be able to combine the values of children that are evaluated by distinct processes, these values are stored in shared memory: the top part of the tree down to, and including, the nodes that are evaluated by worker processes, is built explicitly in shared memory.

The 'leaf' nodes of this explicit tree are evaluated by worker processes using sequential alpha–beta search without building the subtrees explicitly. When one of the workers has evaluated its board position (subtree), it will propagate this new value in the explicitly-constructed tree, so that the next worker to start evaluating a new position will have better alpha and beta bounds.

Both implementations use a fan-out of 38 and a search depth of 6. Only the top two levels of the search tree are built explicitly in shared memory, so that there are 38 subtrees to search.

The measurements were obtained using three different static evaluation functions. We supply the mean values of the measurements for these three functions. The speed-ups for alpha–beta, which are depicted in Figure 7, are clearly less than those for the travelling-salesman problem. This is because, in contrast to the TSP algorithm, a parallel implementation of alpha–beta always searches more nodes than the sequential implementation. This is caused by the fact that at the start of the search, there are no good alpha and beta values available to prune the search tree. In sequential alpha–beta, only the first subtree suffers from these weak alpha and beta values. The next subtree can be pruned with much stronger alpha and beta values (produced by the first subtree), so it can be pruned earlier. In the parallel implementation, each process evaluating a subtree starts with the weak alpha and beta values. On top of that, updating the top of the tree is only useful for jobs that still have to be started. Processes that are already working on a subtree cannot use these new values of their top-node to reduce the search space. Thus the more processes are used, the more nodes are searched relative to sequential alpha–beta.

Matrix multiplication

We parallelized matrix multiplication by dividing the rows of the matrix among the available processes. When the two input matrices are known to each process, each process can compute the results for its rows independently. In the Orca program, each process generates the same (pseudo-random) input matrices locally: they are not shared. The output is produced by printing the result rows in turn. Therefore the result is not shared either. In the SVM-system program, we used the same

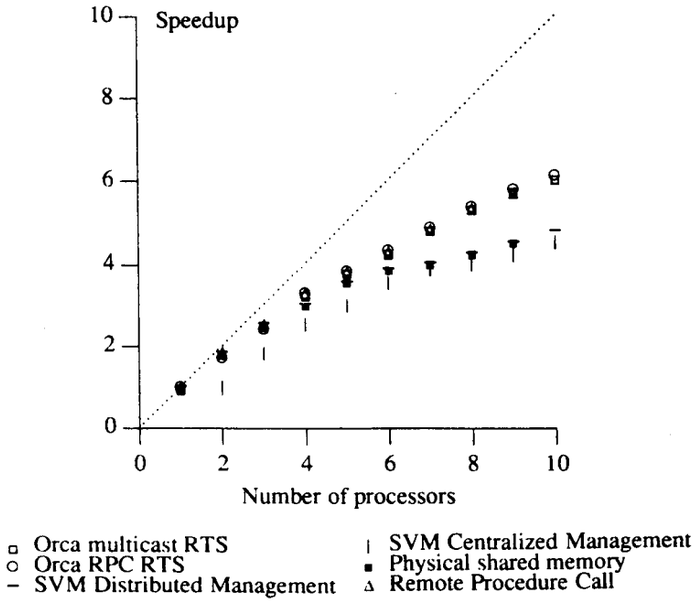

Figure 7. Alpha-beta

approach. Thus both systems only use shared memory for synchronization and work division, not for sharing of input and output.

The algorithm has a constant factor: the generation of the matrices. As the number of processes increases, the speed-ups lag more and more behind. The measurements were obtained by multiplying two 250 × 250 integer matrices. The speed-ups are shown in Figure 8.

All-pairs shortest-paths problem

The all-pairs shortest-paths (ASP) algorithm computes the shortest distance from every node to every other node in a graph. A sequential algorithm is due to Floyd.[26] For an $N \times N$ matrix representing the graph, it uses N iterations to compute the distance matrix. In iteration k, for every node combination $1 \le i, j \le N$:

$$\delta^{k}(i,j) = \min \{ \delta^{k-1}(i,j), \delta^{k-1}(i,k) + \delta^{k-1}(k,j) \}$$

For all k, $\delta^{k}(i,j)$ *is* the length of the shortest path between node i and node j that passes through nodes 1, ... k only. Matrix δ^{N} contains the desired result.

This algorithm can also be parallelized by dividing the rows to compute among the processes. But for a process to compute a row in iteration k, it needs $\delta^{k-1}(k,j)$ $1 \le j \le N$, that is row k of iteration $k-1$. These rows are shared and their availability synchronizes the processes. Speed-ups for a 200 × 200 matrix are shown in Figure 9.

The computation of iteration k can only proceed when row k of iteration $k-1$ is available. During the computations, all processes are waiting for the same data. If

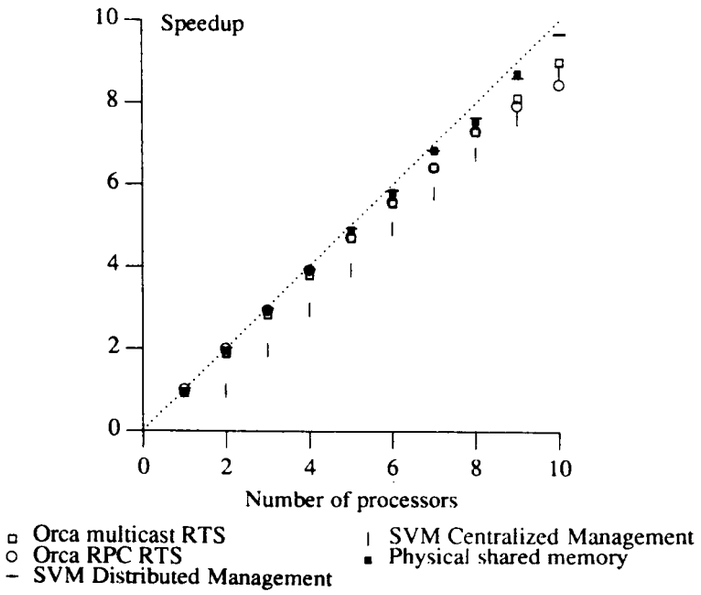

Figure 8. Matrix multiplication

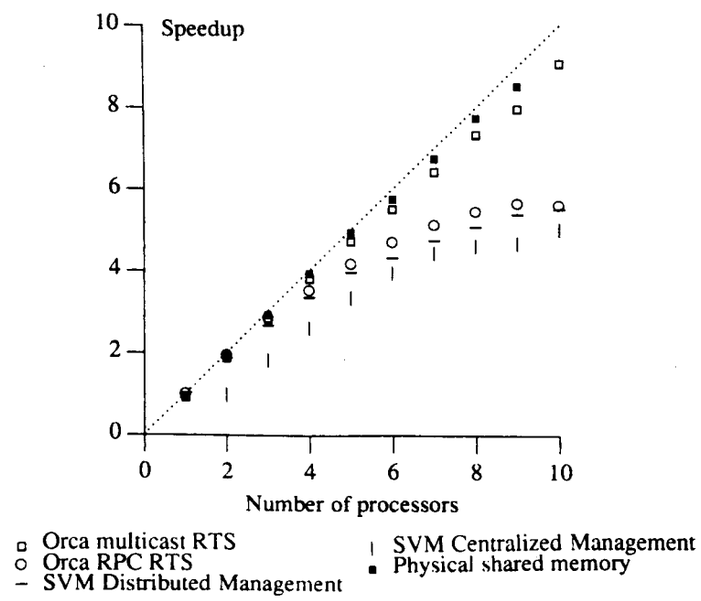

Figure 9. All pairs shortest paths

these data, when they become available, are distributed over the processes one by one, each row becomes a sequential bottleneck. So, the only system (besides physical shared memory) that performs well is the multicast shared data-object system. In this system, the new value is broadcast to all processes in one message.

DISCUSSION

Both the structured and the unstructured systems achieve good speed-ups for most applications. Overall, the speed-ups for Orca programs are better. This can be attributed to three facts.

1. In Orca programs, the granularity of the shared data is inherently tailored to the application. A message only has to be as large as the data it contains. The run-time system can even choose between sending the new value or sending the operation, depending on the costs. In the SVM system, shared data are always transferred in chunks of 1 page. Pages of 4K, as we use, are mostly too big: just a small portion of the page is of interest, the remainder wastes time and bandwidth. Although this does not occur in our application, 4K could be too small and multiple requests have to be sent to get the whole data-structure. What does occasionally happen is that a data-structure (e.g. a row of a matrix) spans a page boundary. To access it, two pages have to be referenced, instead of one.

   This effect of wasted time and bandwidth is reflected in the results for the travelling-salesman problem (TSP) and alpha–beta search. Getting a new job requires a small message in the shared data-object systems, whereas in the SVM systems a message of 4K must be sent. In the TSP problem updating the minimum leads to the same situation.
2. The structured paradigm uses updating whereas the unstructured paradigm uses invalidation to keep the address space coherent. When a process changes some shared data, and these data will be referenced by most other processes before further changes, the update approach performs better. This is the case in all applications except alpha-beta.
3. Because of the overhead of the Orca run-time system, which is still a prototype, sequential Orca programs (except alpha–beta) are three to four times slower than the sequential C programs. Therefore, the communication channel is relatively three to four times faster for the Orca programs than for the C programs; it will not become a bottleneck as quickly as in the unstructured systems. When a better Orca compiler is available, we shall be able to make more realistic measurements.

Multicast RTS versus RPC RTS

For the most communication-intensive application, ASP, the broadcast protocol performs very well. As explained earlier, the distribution of the newly-generated data is no sequential bottleneck. The positive effect of broadcasting a new value versus updating through a two-phase primary copy protocol can also be seen in the TSP measurements. Because the shared variable minimum is read very frequently by all processes, they all have a local copy. Hence the RPC RTS has no advantage of partial replication. A multicast update is more efficient in this situation.

In the alpha–beta algorithm the shared top-part of the tree is only referenced when a job is completed. Before the next reference, this shared variable is updated a few times by other processes. The RPC RTS therefore does not replicate it at every processor, but just maintains a primary copy at one processor. In contrast to the full replication multicast RTS, where updating the tree involves all processors, in the RPC RTS only two processors are involved. This effect can be seen in the results of alpha-beta which is the only application where the RPC RTS is equally fast as the multicast RTS. Which approach performs better depends on the access patterns a specific application exhibits.

Distributed versus centralized management

For all applications the speed-ups in the central manager system keep in step with the speed-ups in the distributed management system. Clearly, for 10 processors, the central manager is not a bottleneck. The measurements for TSP even show better speed-ups for the central manager system than for the distributed management system. This can be explained as follows: when the shared variable minimum is updated, all copies are invalidated. In the central manager system, every process contacts the central manager to get a read-copy again. For the first of these requests, the central manager has to contact the new owner with an RPC. But all other requests can be handled without an RPC to the owner, because the central manager has a valid copy available. Therefore these requests need one RPC only.

In the distributed management system, each process broadcasts a request and waits for the page to arrive, so a broadcast and an RPC are needed. Furthermore, the owner process, which could otherwise work on the application, is interrupted to send the page. In this way not only the requester, but also the current owner of the page, lose computation time. Lastly, each broadcast is received by all processors, which means that each processor is interrupted at every broadcast, even though a given processor can ignore most broadcasts. The main properties that affect performance are summarized in Table II.

Good versus bad placement in the SVM systems

The difference in performance between SVM programs where the shared variables are cleverly distributed over the pages, and programs where they are not, can only be seen in the TSP program. The other applications do not suffer from poor placement for the following reasons. Matrix multiplication does not do much com-

Table II. The four most important properties that affect performance

	Centralized management	Distributed management	RPC RTS	Multicast RTS
Unit of sharing	Page	Page	Data-object	Data-object
Replication	Partial	Partial	Partial	Full
Invalidation	Yes, one by one	Yes, broadcast	No	No
Updating	On demand, one by one	On demand, one by one	Always, one by one	Always, multicast

munication at all. The ASP program communicates through a shared matrix occupying 40 pages, where only the first five rows of the matrix are positioned on a page containing other shared data. In alpha–beta, once a process gets a job, it will not reference shared memory further until it has finished that job completely. References to shared memory are therefore infrequent, and pages being paged out do not delay the computation.

Only with the TSP problem do the speed-ups lag when the data are poorly distributed. This is explained by the fact that every time a process takes a new job, a counter in shared memory must be updated and the shared-variable minimum, which lies on the same page, is inaccessible for all other processes. These processes cannot continue their computation until the page is paged in again. The effect on the speed-ups can be seen in Figure 10.

Programmability aspects

In general, the programming of the structured DSM model is much easier, for the following reasons:

1. Orca provides type security. This means that incorrect use of variables is detected, either by the compiler or by the RTS. For instance: assigning an integer value to a boolean variable, or supplying an out of range index to an array. Because parallel programs are mostly non-deterministic they are difficult to debug. The extra support type security provides is very useful. The type security offered by Orca is very helpful with debugging.
2. The implicit mutual-exclusion synchronization, supplied with operations on shared data-objects, makes the model easy to understand and less burdensome to program than the SVM system, where most accesses to shared memory must be explicitly protected by locks. Another advantage of implicit mutual-exclusion synchronization is that programs are more compact and easier to read.

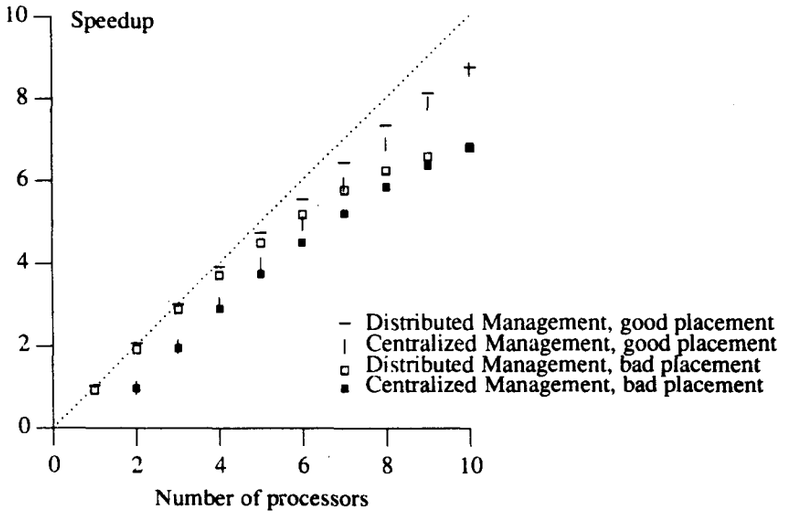

Figure 10. Travelling-salesman problem with good and bad placement

3. In the SVM system, the programmer can only use the shared memory efficiently when he or she understands (at least part) of the working of the system. The placement of variables is especially important. Another consideration is what lock to use: a read or a write-lock, or possibly no lock at all. Because the shared data-object model provides a structured approach to DSM, the programmer is shielded from the low-level functionality of the system. The system itself takes care of efficient replication of objects.

Scalability

To achieve the goal of high performance at low cost one needs to know if the applications will scale to a larger number of processors than 10. There are a number of factors that influence the scalability of both DSM models: the access patterns of the applications to DSM, the granularity of the operations, and the size of the messages that are sent to keep DSM consistent. Although we cannot make any hard statements about these factors, we shall discuss the influence on scalability of each of them. Our assumption is that if network traffic can be avoided, applications will scale well.

The access patterns that an application exhibits determine how well the underlying DSM model will scale. Applications that perform a few write operations but perform many read operations will scale to a large number of processors, because they will not generate any network traffic due to the replication schemes that DSM systems use. This can already be seen at 10 CPUs. For TSP, the physical shared memory system performs much worse than any of the DSM implementations, because it does not have any hardware scheme to replicate shared memory.

The second important factor is the granularity of the operations. If operations of the DSM are low-level operations, such as move a word from a register to main memory', these operations can lead to a large amount of network traffic. Consider the case that a shared record is updated continuously by two processors. In an unstructured DSM implementation this will generate many messages, if the programmer does not lock the record explicitly. It might happen that for each write on a word of the record, a message has to be sent. In a structured DSM implementation the operation on the record is packed in one user-defined procedure call. In this case, only one message has to be sent.

The third important factor is the size of the messages that are sent. In an unstructured DSM, each time a write operation on a page that is not located on the local processor has to be performed, the DSM system has to get the complete page across the network, even if the write required only four bytes of the page to be changed. Furthermore, if such a four-byte object crosses page boundaries, two complete pages have to be sent over. In a structured DSM, the RTS only sends over the operation code or moves the object. In both cases, typically only a limited number of bytes has to be transferred.

Both unstructured and structured DSM rely on replication schemes and dynamic placing of shared memory to scale to a large number of processors. However, the unstructured DSM will in general need more messages of a larger size than a structured DSM to keep shared memory consistent. In general, one might expect that structured DSM would scale better than unstructured DSM. Ultimately, however, scaling will be determined by the application.

CONCLUSIONS

In this paper we have compared two distributed shared memory paradigms, one structured and one unstructured. This comparison was mainly done on the basis of measured execution times for four parallel applications. The unstructured shared virtual memory paradigm showed better absolute execution times, whereas the structured shared data-object paradigm showed better speed-ups. A more efficient implementation for the Orca RTS than the current prototype could make this difference in absolute times smaller. It remains to be seen if the speed-ups for such a faster RTS will stay at their current high level. However, it is our expectation that they will be better than the speed-ups for the SVM system, because of the structured application-dependent division of shared memory.

In terms of programmability and readability, programming applications for the shared data-object model is clearly superior to programming applications for the shared virtual memory model; the clean semantics of the shared data make it easy to write and debug programs. In addition, the programs execute efficiently, even for users who do not understand the underlying system.

An advantage of the SVM system is that every programming language supporting global data could be extended with a few system calls to make use of distributed shared memory. Implementing the shared data-object model within an existing language is less easy.

In conclusion, both paradigms are useful for distributed programming and can be implemented efficiently. Although the SVM paradigm currently has the faster implementation, the shared data-object paradigm is easier to understand, to program and to debug. On top of that, it is more promising in terms of performance, as soon as a better Orca compiler is available.

APPENDIX

This appendix lists the execution times for our four applications. The tables contain the mean values over a number of runs. All times are in seconds. The measurements for the SVM Central Manager system start at two processors because there is always one processor occupied by the Central Manager. We have no measurements with 10 processors for the physical shared memory system because we only had nine working processors.

Table III. Travelling-salesman problem (execution time in seconds)

No. processors	1	2	3	4	5	6	7	8	9	1	0
SVM, centralized, bad placement		166·0	84·0	56·7	44·1	36·6	31·7	28·3	26·0	24·1	
SVM, distributed, bad placement	166·3	83·4	56·4	44·0	36·4	31·7	28·5	~6·4	25·0	24·2	
SVM, centralized, good placement		159·6	79·8	52·7	39·8	32·2	27·0	23·2	20·2	18·3	
SVM, distributed, good placement	159·9	78·7	53·8	41·1	34·0	29·0	25·0	21·9	19·8	18·3	
Orca multicast	587·7	291·7	192·6	144	·7	116·6	97·3	83·7	73·2	65·8	59·1
Orca RPC	660·8	327·7	217·1	164	·2	132	·3	111	·3	95·6	84·4	76·1	69·0
Shared memory	169·6	84·6	55·8	42·2	33·9	28·2	24·2	21·2	18·9		
Amoeba RPC	145·6	75·6	52·2	41·5	35·1	31·0	28·4	26·8	24·5		

Table IV. Alpha-beta search (execution times in seconds)

No. processors	1	2	3	4	5	6	7	8	9	1	0
SVM, centralized, bad placement		1159.4	634.9	457.1	3854	326.4	299	.0289.2	273.7	256.9	
SVM, distributed, bad placement	1160·5	635·4	458 ·0	386·8	327·6	300·0	290	·3274·6	258·0	243·3	
SVM, centralized, good placement		1159·2	634 ·7	457·0	385·7	326·2	299	·0289·1	273·8	256·7	
SVM, distributed, good placement	1160·5	635·1	458 ·1	386·8	327·5	300·2	290·	4274·6	258·2	243·5	
Orca multicast	1801·5	972·1	713 ·4	547·5	470·8	418·5	371·	0336 ·0	311·6	296·4	
Orca RPC	1816·1	1063·7	752 ·6	551·8	474·5	421·8	373·	9338·7	3141	2966	
Shared memory	970·6	530·7	382 ·2	322·4	272·9	249·9	241	·7228·6	214·8		
Amoeba RPC	1813·4	978·5	718 ·3	552·2	475·0	424·5	377	·1342·3	319·2		

Table V. Matrix multiplication (execution times in seconds)

No. processors	1	2	3	4	5	6	7	8	9	1	0
SVM, centralized, bad placement		170·8	85·5	57·5	43·6	34·5	29·2	25·2	22·3	200	
SVM, distributed, bad placement	171·0	86·0	57·7	43·5	34·7	29·1	25·2	22·1	19·6	17·8	
SVM, centralized, good placement		170·8	85·9	57·6	43·5	34·5	29·3	25·2	22·3	19·5	
SVM, distributed, good placement	171·0	85·8	58·0	43·7	35·0	29·3	25·1	22·4	19·9	17·7	
Orca multicast	810·3	410·6	279·1	209·8	169·9	143·8	124·2	109·8	98·7	89·1	
Orca RPC	780·1	392·9	265·8	200·0	166·0	139·8	121·6	106·5	98·2	92·3	
Shared memory	180·5	90·5	60·8	45·9	36·6	30·8	26·2	23·7	20·6		

Table VI. All pairs shortest paths problem (execution times in seconds)

No. processors	1	2	3	4	5	6	7	8	9	1	0
SVM, centralized, bad placement		68·8	37·1	26·0	20·8	17·2	16·1	14·6	14·7	13·0	
SVM, distributed, bad placement	68·6	37·0	26·0	20·6	17·6	15·6	15·0	13·5	12·6	12·7	
SVM, centralized, good placement		68·6	37·1	26·1	20·4	17·3	15·4	14·8	14·6	13·6	
SVM, distributed, good placement	68·7	37·0	26·0	20·7	17·5	16·0	14·6	13·6	12·8	12·5	
Orca multicast	432·1	218·9	148·0	111·4	90·0	77·1	66·3	58·2	53·6	47·1	
Orca RPC	400·0	204·7	140·0	114·1	95·9	85·1	78·2	73·2	70·5	71·2	
Shared memory	68·6	34·4	23·1	17·3	13·8	11·8	10·1	8·8	8·0		

REFERENCES

1. K. Li, 'Shared virtual memory on loosely coupled multiprocessors', *Research Report 492 (Ph. D. Dissertation)*, Yale University, New Haven, CT, September 1986.
2. K. Li and P. Hudak, 'Memory coherence in shared virtual memory systems', *ACM Trans. Comp Syst.* , 7, (4), 321–359 (1989).
3. H. E. Bal, M. F. Kaashoek and A.S. Tanenbaum, 'Orca: a language for parallel programming of distributed systems', *IEEE Trans. Software Engineering*, **18,** (3), 190–205 (1992).
4. K. Li, 'IVY: a shared virtual memory system for parallel computing', *Proc. 1988 International Conference Parallel Processing*, St. Charles, IL, August 1988, Vol II, pp. 94–101.
5. K. Li and R. Schaefer, 'A hypercube shared virtual memory system', *Proc.* 1989 *International Conference Parallel Processing*, St. Charles, IL, August 1989, Vol I, pp. 125–132.
6. B.D. Fleisch and G. J. Popek, 'Mirage: a coherent distributed shared memory design', *Proc.*

Twelfth ACM Symposium on Operating Systems Principles, Litchfield Park, AZ, December, 1989, pp. 211–223.
7. R. G. Minnich and D. J. Farber, 'The Mether system: distributed shared memory for SunOS 4.0', *USENIX Summer '89,* 1989, pp. 51–60.
8. R. G. Minnich and D. J. Farber, 'Reducing host load, network contention, and latency in a distributed shared memory system', *Proc. Tenth International Conference on Distributed Computing Systems,* Paris, May 1990, pp. 468–475.
9. J. K. Bennet, J. B. Carter and W. Zwaenepoel, 'Munin: distributed shared memory based on type-specific memory coherence *'Proc. Second Symposium on Principles and Practice of Parallel Programming,* Seattle, WA, March 1990, pp. 168–176.
10. R. Bisiani and A. Forin, 'Architectural support for multilanguage parallel programming on heterogeneous systems', *Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, CA, October 1987, pp. 21–30.
11. S. Ahuja, N. Carriero and D. Gelernter, 'Linda and friends', *IEEE Computer,* **19,** (8), 26–34 (1986).
12. David Gelernter, 'Generative communication in Linda', *ACM Trans. Prog. Lang. Syst.,* **7,** (l), 80–112 (1985).
13. S. E. Lucco, 'Parallel programming in a virtual object space', *SIGPLAN Notices (Proc. Object-Oriented Programming Systems, Languages and Applications,* Orlando, FL, 1987), **22,** (12), 26–34 (1987).
14. H. E. Bal and A. S. Tanenbaum, 'Distributed programming with shared data', *Computer Languages,* **16,** (2), 129–146 (1991).
15. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen and G. van Rossum, 'Experiences with the Amoeba distributed operating system', *Commun. ACM,* **33,** (12), 46–63 (1990).
16. M. F. Kaashoek, A. S. Tanenbaum, S. Flynn Hummel and H. E. Bal, 'An efficient reliable broadcast protocol', *Operating Systems Review,* **23,** (4), 5–20 (1989).
17. M. F. Kaashoek and A. S. Tanenbaum, 'Group communication in the Amoeba distributed operating system', *11th Int. Conf. on Distributed Computing Systems,* Arlington, Texas, 20–24 May 1991, pp. 222–230.
18. H. E. Bal, *Programming Distributed Systems,* Silicon Press, Summit, NJ, 1990.
19. H. E. Bal, M. F. Kaashoek and A. S. Tanenbaum, 'Experience with distributed programming in Orca', *Proc. IEEE CS 1990 Int. Conf. on Computer Languages,* New Orleans, LA, March 1990, pp. 79–89.
20. A. S. Tanenbaum, M. F. Kaashoek and H. E. Bal, 'Parallel programming using shared objects and broadcasting', *IEEE Computer,* August 1992.
21. K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, 'The notion of consistency and predicate locks in a database system', *Commun. ACM,* **19,** (11), 624–633 (1978).
22. H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum and J. Jansen, 'Replication techniques for speeding up parallel applications on distributed systems', *Concurrency: Practice and Experience,* **4,** 337–355 (1992).
23. B. Liskov and L. Shrira, 'Promises: linguistic support for efficient asynchronous procedure calls in distributed systems', *Proc. SIGPLAN 88 Conf. on Progr. Lang. Design and Impl.,* Atlanta, GA, June 1988, pp. 260–267.
24. H. E. Bal, R. van Renesse and A. S. Tanenbaum, 'Implementing distributed algorithms using remote procedure calls', *Proc. AFIPS Nat. Computer Conf.,* Chicago, IL, June 1987, Vol. 56, pp. 499–506, AFIPS Press.
25. D. E. Knuth and R. W. Moore, 'An analysis of alpha–beta pruning', *Artificial Intelligence,* **6,** 293–326 (1975).
26. R. W. Floyd, 'Algorithm 97: shortest path', *Commun. ACM,* **5,** 345 (1962).