CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING
CONGRES **CANADIEN** EN GENIE ELECTRIQUE ET INFORMATIQUE
Montreal PQ, Canada, September **17-20** September 1989

Shared Virtual Memory: A Simple Model for Implementing Distributed Applications

S. Zhou,T. McInerney, M. Snelgrove.M. Stumm,D. Wortman
Computer Systems Research Institute
University of Toronto
Toronto, Ontario M5S 1A4

Abstract

We first describe the design and implementation of a distributed shared memory system for a cluster of Sun workstations We then present performance results for a simple parallel application and show that distributed shared memory is competitive with direct use of message passing The factors affecting SVM performance are also studied. The rest of the paper focuses on our current research into extending the shared memory model to ***heterogeneous systems,*** which is believed to offer substantial performance benefits as well as a number of challenging problems. We identify some of these problems, and propose solutions to one of them.

1 Introduction

As the microclcctronic technology advances rapidly, the structure of computer systems has shifted from isolated machines supporting time sharing to distributed environments with many personal and server nodes connected by communications networks. The dramatic increases of the total computing resources, such as CPU cycles, memory space, and I/O bandwidth, suggest substantial potential for computing capabilities approaching or even exceeding those available on supercomputers. However, such resources are often under-utilized because they are distributed on a number of nodes with their own operating systems managing their local resources.

One method to better utilize the computing resources to yield higher applications performance (measured, for example, by job response time) is to run parts of the same application simultaneously on different nodes. Unfortunately, two problems have hindered the wide spread use of such distributed applications. First, the programming of such applicntions has been substantially more difficult than that of traditional sequential applications. Not only will a problem have to be partitioned, but also the communication and data sharing among the modules running on different nodes will have to be facilitated to satisfy various data ***and*** control dependencies intrinsic to the application. Secondly, the nodes in a distributed system are, more often than not, different, in terms of their operating systerns and/or hnrdwnre architectures. Such system heterogeneity makes the communication among the modules more difficult, as data conversion and common communication protocol are required.

Three methods of communication and dnta sharing for distributed applicntions have been proposed and implcmentcd. The earliest one used is message passing. .4 message transport mechanism is provided by the system, and all message packaging and interpretation work is done by the particular application. Though offering the greatest flexibility and performance, message passing places a great deal of responsibility on the application writer. Not only does he/she have to understand the problem at hand, find a way to partition it, and implement the relnted algorithms, but he/she also has to understand the communication protocol details of the system, and specify the message interactions among the modules.

Remote procedure calls (RPC) are an attempt to shield the programmer from the transport level details of communication [l]. A model similar to conventional procedure calls is used to specify interactions between distributed modules. The code to pack/unpack the underlying messages and to activate the associated operations is either provided by the system, or automatically generated from a specification file. However, data sharing among the distributed modules is still explicit, and RPC's ability in dealing with heterogeneity is limited. Like ordinary procedure calls the basic RPC is synchronous, hence cannot support parallel execution of modules, though extensions for such purposes have been proposed (for example, see [6]).

A third technique to support communication and data sharing in a distributed application is ***shared virtual memory (SVM) [3] [4].*** In SVM, a virtual address space is maintained on all the hosts running modules of an application. Pages of this address space can be brought to and kept on the hosts as they are needed, and their contents are kept consistent implicitly by the system. To the user, the system appears to be a shared memory multiprocessor: Any memory location in the shared address space can be accessed by any participating host at any time, and the most recent value at that location is always returned. Thus, no explicit message passing and data sharing needs to be programmed for an application. Compared to message passing and RPC, SVM offers a simpler programming environment, and makes applications more portable.

In this paper, we present an SVM system implemented on a cluster of Sun workstations. Our objectives in building the system are to study the design and implementation issues involved, to experiment with writing distributed applications for such systems and evaluate their performance and to extend the shared virtual memory model to heterogeneous environments. Our SVM is described in Section 2. Some performance measurements ***of*** a distributed application implemented in the SVM system are presented in Section 3. In Section 4, we discuss our current work to extend the SVM model to heterogeneous environments. Finally, some concluding remarks will be made in Section 5.

2 Design and Implementation

The central part of our SVM system is the virtual memory management that provides the illusion of a consistent address space across machine boundaries. A thread mechanism is available to allow multiple streams of execution to share the same address space and to migrate among the nodes The specific communication needs of the virtual memory management and the thread management are supported by a communication module implementing a request-response protocol, and built on top of the system-provided UDP/IP protocol. The request-response paradigm for communication is suitable for SVM to support page requests and thread interactions. In this section, we discuss the design and implementation of our SVM system in SunOS [7], focusing primarily on the above components.

2.1 Virtual Memory Management

Virtual memory management performs two major functions: allocation of chunks of memory in the shared address space, and maintenance of the page-level consistency of the space. We discuss each in turn. each to memory allocation in a traditional virtual address space, memory in a shared virtual space is allocated by sending requests to the allocation ***manager.*** Since such requests may bc issued from any node, a centralized manager is needed to avoid granting multiple requests for the same memory region. Although it is possible to have multiple allocation managers each responsible for a separate range of the space such flexibility was deemed to be unnecessary, as it complicates the 'allocation operations, which are typically infrequent anyway. For example, if one of the managers does not have enough space left to grant a request, either another manager would have to be contacted, or a failure returned unnecessarily. In our system, we employed ***a*** single allocation manager that gives out virtual memory using a simple sequential allocation algorithm.

Independent from the allocation manager, each page in the ***space IS*** associatecl with a consistency manager, and an owner The manager of a page keeps track of the owner of the page, and the set of nodes having copies of that page, together with their access rights. There are three types of access rights: nil, read, and write. A conventional multiple-reader-single-writer page consistency policy is used: When a page fault for a shared page occurs, the handling routine sends a page request (for either read or write) to the manager of the page, which forwards it to the owner. If it is a read request and the owner has (non-exclusive) read privilege, a copy of the page is sent to the requesting node, and the requesting node is added to the copy set kept at the manager node. If, on the other hand, the owner has (exclusive) vrite access, it will give out a copy, and change its own access to read. If the page fault results from a write access, the faulting node needs exclusive write access to that page. The owner node's access right is changed to nil, and all the other copies of the page are invalidated. The page's ownership is then sent to the requesting node, along with the page. For all the above operations, the manager node is responsible for initiating the actions upon receiving the request. For the special cases in which some or all of the three parties - the requester, the manager, and the owner - reside on the same node, the same protocol is used, except that some messages become unnecessary.

From the above description of the consistency algorithm, it should be clear that only the ***manager*** needs to know the owner of the page. However, it is necessary for every node to be able to determine the manager of each page in the shared space efficiently. The simplest method to do this would be to have ***a*** centralized manager for all pages. However, both intuition and experiments show that the central manager tends to become the bottleneck of the system, adversely affecting the performance of the application. A better method would be to use fixed ***distributed managers,*** where the ***manager of*** a page is determined_ by applying a simple hash function to the page number. We adopted such a method in our Sun implementation. It is possible to eliminate the communication between the manager and the owner by making both reside on the same node. In this case, however, it is harder to determine the location of the manager, as it changes with the owner.

One potential performance problem of the above consistency algorithm is that of page ***thrashing among*** the nodes, much like virtual memory thrashing between main memory and the backup store. Page thrashing can be serious if data frequently updated by several nodes is located in the same page. One way to alleviate such thrashing is to make a page "stick" to a node for a while before allowing it to move to another node. Another method is to avoid allocating to the same page data that will be concurrently updated by multiple nodes. This requires knowledge of the application, and must therefore be performed by the application programmer.

2.2 Thread Management

Light weight processes, or threads, are natural companions of shared virtual memory, as multipIe threads share an address space. Supporting threads is very useful for programming in SVM as well. For instance, several threads can be created on one node, have their computational tasks set up, and then moved to other nodes for execution. Though some more recent operating systems provide support for threads [2] [8], most systems do not. For our implementation in SunOS, a thread package has been implemented as part of the SVM. This includes the creation, migration, .as wel1 as scheduling of threads. A number of synchronization primitives are also provided. Each thread has a control block, and a stack allocated in the shared address space. Dynamic thread migration is realized by transferring the control block, and having the stack faulted in on demand.

2.3 A Few Implementation Issues

When building an SVM system on an existing operating system, there are two basic choices for implementation. One is to integrate SVM into the operating system kernel, while the other is to implement SVM at the user level as a library package linked into applications. The basic tradeoff is performance versus flesibility. We chose to have a user-level implementation. The thread management, SVM page table management, and communication modules exist within the context of one conventional UNIX process on each participating node. A few changes to the SunOS, however, have been made. First, in order to handle SVM page faults at the user level, the memory management of SunOS was modified, to allow the setting of page access rights, and to register a user Level routine as the fault handling routine If no page access fault occurs when an SVM page is accessed, the native memory management performs the necessary translation, with attendant good performance. In the case ***of an SVM*** page access 'fault (i.e., nonexistent page, or write access on a read-only page), the user-level handling routine obtains control, and follows the consistency protocol described in Section 2.1. Memory protection for that page is then changed so that subsequent accesses **to** that page will be handled by the native MMU hardware.

Another change we made to SunOS is to detect a read page fault immediateiy followed by a write fault, so the read fault can be avoided. It is frequently the case that before a data item is updated, its value is read first, resulting in an unnecessary read fault. The change to the page fault detection allows us to trigger write fault directly, saving an additional read fault. This is intended as ***a*** performance enhancement.

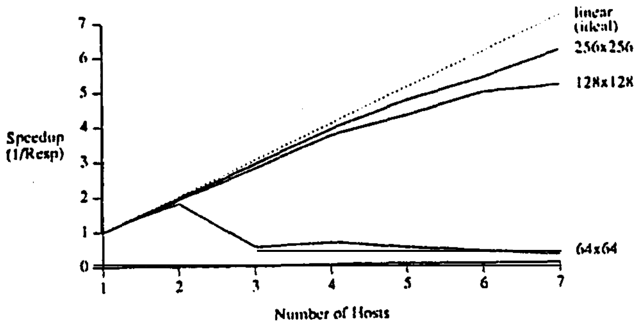

Figure 1: Speed up *of* Matrix Multiplication

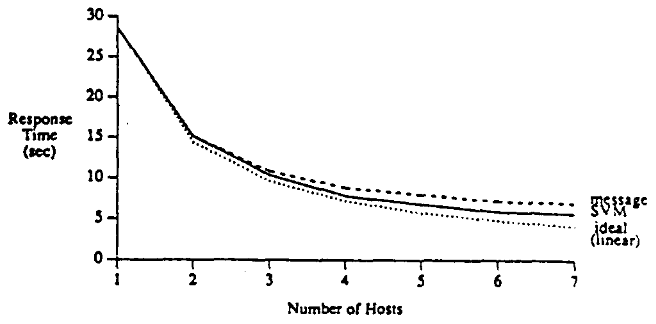

Figure 2: SVM and Message Passing Comparison (128x128 matrix)

3 Performance Evaluation

We have implemented a number of parallel and distributed applications on our SVM system on SunOS. In this section, we consider one of them, namely parallel matrix multiplication, in detail, This is a relatively simple, but very frequently used type of computation. One natural parallel implementation is to distribute the computation of the rows for an $N \times N$ square result matrix to the P available processors, and to collect the results to form the complete matrix. We varied the values of N and P to study their effects, and compared the performance *of* our SVM implementation to that of one using message passing. All of our experiments were performed on a cluster of Sun 3/60 diskless workstations.

3.1 Problem Size and Scalability

Figure 1 shows the speedup factors achieved on various numbers *of* hosts using our SVM implementation, and for matrices of different sizes. It can be seen that, for larger problem sizes, the speedup is close to linear up to at least seven hosts. when problem size is small (e,g., 64x64), however, there is hardly any speedup at all; applying more hosts to the problem may even slow down the computation. This is to be expected, as small matrices fit in one or very few pages; updating the elements on different hosts causes page transfers (or even thrashing) that overshadows the benefits of the small amounts of parallel corn-putation. For realistic application of parallel computations, the problem sizes are usually large as small problems do not take much time to execute (for example, a 64x64 matrix multiplication implemented sequentially and running on a single Sun3/60 takes only 3.6 seconds), and are therefore not of much interest.

A crucial factor in SVM performance *is* page contention. For matrix multiplication, this translates into page alignment. If a page in the result matrix is shared by two hosts, and they update elements in turn, page thrashing occurs. The performance impact depends on the particular timing condition. This was observed *for* the small matrices, where the response time fluctuates widely from run to run. For large matrices, on the other hand, a smaller percentage of pages are shared, so response time becomes very stable, and is linear in number of hosts, Performance can be further improved by partitioning the problem along page boundaries, rather than into sizes as equal as possible.

For different matrix sizes beyond 128x128, we observed that response time grows as the cube *of* the matrix dimension, For instance, a 256x256 parallel matrix multiplication takes about eight times as long to compute as a 128x128 multiplication. This agrees with the growth of the intrinsic amount of computation (scalar multiplication and addition), and indicates that SVM overhead remains stable over large problem sizes,

3.2 Comparison to Message Passing

To assess the SVM overhead, we compared the response time of our SVM implementation to that of message passing, using the same multiplication algorithm. The results are shown in Figure 2. The performance difference between SVM and message passing is small. When the number of hosts is three or over, we even observed better performance for SVM. This is probably because the transfers of pages in SVM are performed on demand! parallel to computations on the hosts, whereas for the message passing method, the data transfers need to be performed explicitly and synchronously. Similar observation was made for the 256x256 case.

3.3 Programming Efforts

Besides the above measures of quantitative performance, the efforts in programming *a* parallel application is extremely important, and should not be overlooked. Since communication and data sharing is hidden from the application in SVM, the programmer can concentrate on the the problem at hand, rather than spending time planning data transfers, ensuring data consistency, and deciding which transport protocol to use. Consequently, the programming effort is much reduced, and cleaner and more portable code results. In light of the minimal overhead for sufficiently large problems, we believe that the SVM paradigm is preferable to message passing for many types of applications.

4 Tackling Heterogeneity

Being able to accommodate heterogeneous systems environment can substantially expand the application domain of the SVM 'paradigm. Modern computing environments are increasingly of heterogeneous nature, encompassing workstations, shared server hosts, and possibly multiprocessor and mainframe machines. These machines are likely different in their architectures and/or operating systems. SVM presents a simple mechanism to mask out these differences, while allowing applications to take advantage of the computational resources in the system. For example, an application can be developed and started on a workstation with good user *interface* support, and major parts of its computation be distributed onto the much more powerful server hosts, completely transparently.

Like heterogeneous RPC, SVM in heterogeneous systems presents a number of difficult problems, the study of which is a major thrust of our current research [5]. First, the page sizes may be different on the machines, making the consistency protocol more complicated. Secondly, the format for data representation may be different. For example, the byte order and/or the sizes of certain data types may differ. When sharing pages, some data conversion will be unavoidable. Knowledge about the content of pages is needed. Lastly, different executable code has to be generated for each type *of* machine and/or operating system, and, if dynamic thread migration is to be supported, equivalent execution points, or ***migratable*** points, will have to be identified. We believe the last problem is less critical, and, hence, will concentrate on the first two issues in the following discussion.

4.1 Page Sizes

A simple algorithm for handling different page sizes is to use the largest ***page*** size ***among*** the architectures as the granularity for the memory consistency protocol. Since page sizes are powers of two in bytes, smaller pages do not cross boundaries of large pages. For those machines with small pages, the pages are grouped, so a page fault will result in multiple pages being transferred, and their protections being changed accordingly. For example, we are implementing a heterogeneous SVM (HSVM) system between DEC's experimental Firefly multiprocessors [8] and Sun workstations. Firefly's page size is 1 KB[1], whereas Sun's page size is S KB. Using the simple algorithm above, 8 Firefly pages would be treated as a single unit for consistency purposes.

An obvious problem with the above algorithm is that it unnecessarily increases paging traffic and the possibility of page thrashing due to concurrent write sharing. Another algorithm is ***to*** use smaller page sizes as much as possible. The ***memory consistency*** manager uses the smallest page size as its unit. When a fault occurs on a machine with a larger page size, all the small pages falling in the faulting (large) page will be transferred. On the other hand, for a fault on a machine with the smallest page size, only that page is transferred, and, if the owner of the page also uses this size, only the copy set and possibly the owner of this page need to be changed. For the frequent case of two page sizes, a simple implementation is achievable, and is being planned. We intend to compare the performance of the two algorithms under different classes of applications.

4.2 Data Conversion

Differences in data representations for the heterogeneous machine types may range from byte ordering to different array and record structures. Since pages ***of*** data (but not instruction) need to be shared 'among the machines, data conversion is unavoidable. In the simplest case, where only a range of bytes is shared, nothing needs to be done. Conversion **of** basic types such as inte***ger*** and real numbers may be implemented efficiently. However, for complex data structures, especially those involving pointers, the conversion process may involve significant overhead. In particular the system may be brought to a halt if page thrashing occurs **at** the same time, as expensive data conversion is performed with very little benefit.

In general, complete knowledge about the content of a page is necessary in order to convert it. Such knowledge may be derived during compilation, and be provided to the conversion routine at run time. Otherwise, the programmer must be able to specify the page layout, and convey that information to the conversion routines. Care must be taken to keep the converted data within the same page space, and to convert those data items overlapping page boundaries properly. We are currently investigating data conversion problems in the context of ***our*** Sun-Firefly HSVM implementation.

[1] **Like other machines based on microVAX architecture, Firefly's hardware page** size is 512 bytes. However, the **storage** unit **handled** by virtual memory **is 1KB**, which *is* what *we* **are** concerned with

5 Concluding Remarks

We described the design and implementation of our SVM system. Our experience has confirmed the usefulness of SVM reported by Li and others. The performance results we obtained are quite encouraging for the SVM method, at least for one class of applications. The observations shed light on some of the important factors affecting the performance of distributed applications. We are currently doing similar studies for problems in various application domains, in order to gain more knowledge in this area, and to characterize those types of applications suitable for SVM implementation.

We are extending the SVM paradigm to heterogeneous systems environments Our main research vehicle is an heterogeneous SVM system on **Sun** workstations and Firefly multiprocessors. The page size and data conversion problems discussed in Section 4 will be studied using implementation and measurements.

6 Acknowledgments

We are grateful to Kai Li for his contribution to this project during his four month leave at the University of Toronto. The research discussed herein is partially supported by a grant from the Information Technology Research Center of Ontario.

References

[1] A. Birrell and B. Nelson, "Implementing Remote Procedure Call," ACM Trans. on Computer Systems, 2, 1, February 1984.

[2] D. Cheriton, "The V Distributed System", Comm. ACM 31,3, March 1988.

[3] K. Li, "Shared Virtual Memory on Loosely-coupled Multiprocessors," PhD thesis, Yale University, October 19S6.

[4] K, Li, "IVY: A Shared-Virtual Memory System for Parallel Computing," ***Proc. Inter. Conf. on Parallel*** Processing August 1988.

[5] K Li, M. Stumm, D. Wortman, and S. Zhou, "Shared Virtual Memory Accommodating heterogeneity," Tech. Report CSRI-220, Computer Systems Research Institute! University ***of*** Toronto, December 1988.

[6] J. Sanislo and 'M. Squillante, "An RPC/LWP System for Interconnecting heterogeneous Systems," Proc. USENIX Winter Conference, February 1988.

[7] Sun Microsystems Inc., "Sun-3 Architecture,". August 1986

[8] C. Thacker and L. Stewart, "Firefly: a Multiprocessor Workstation," ***Proc. Second Inter. Conf. on Arch. Support for Prog. Languages and*** Operating ***Systems,*** October 1987