On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers

Federico Meza, Alvaro E. Campos, and Cristian Ruz

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 306 - Santiago 22 - CHILE
[fmeza,acampos,cruz]@ing.puc.cl

Abstract. Distributed shared memory systems provide an easy-to-program parallel environment, to harness the available computing power of PC networks. We present a layered architecture that allows a portable, scalable, and low-cost implementation that runs on Linux and Windows. Only a few, low-level, modules are operating-system dependent; synchronization, distributed memory and consistency management, as well as multithreading are mostly independent. Preliminary results are encouraging; the Linux port performed well, showing high efficiency.

Keywords. Distributed shared memory, parallel programming, parallel systems, distributed systems, multicomputers.

1 Introduction

Parallel computing aims to reduce execution time for applications involving high computational requirements. This reduction is accomplished by distributing work among different processing units that operate in a simultaneous but coordinated way. In a multiprocessor, processing units have access to a common memory; processes communicate through read and write operations on the shared memory. On the other hand, multicomputers have a distributed memory, and communication is achieved by sending and receiving messages; programs request the shared data they need and send the data requested to them.

Multiprocessors are expensive and offer low scalability, but it is easy to write programs for them due to their simple semantics for sharing data. Multicomputers are built using conventional hardware and show better scalability, but communication is slow and programming is harder, since programs must partition the shared data and manage communication between the distributed memories.

Distributed Shared Memory (DSM) systems provide the scalability and low cost of a multicomputer, and the ease of programming of a shared-memory multiprocessor [1]. They offer a virtual shared memory space on top of a distributed-memory multicomputer. This virtual space enables programs on different computers to share memory, even though the computers physically do not share memory at all. All the communication involved is accomplished by the underlying DSM system. Despite the overhead introduced by the DSM layer, applications still perform within acceptable bounds [2].

We are interested in DSM systems that are implemented exclusively at the software level, excluding those systems that require special network or caching hardware. Some software DSM systems may require modifying the compiler to generate sharing and coherence code, specially variable-based and object-based systems. These systems are not considered in this study because they lack portability. We focus on systems supported exclusively by user-level software, using the virtual-memory management primitives of the operating system [3]. As a consequence, sharing granularity is handled at the page level.

In this paper, we outline the key issues involved in the design and implementation of a portable DSM system that runs on top of Linux and Windows. Special attention is given to the operating-system-independent aspects of the design. We claim that using a layered architecture it is possible to reduce the operating-system dependency to a minimum. We present `DSM-PEPE`, a DSM system aimed to use low-cost hardware and conventional networks to provide a distributed parallel environment. Our system conforms a scalable, low-cost, distributed parallel machine that executes programs with shared-memory semantics. A few modules had to be rewritten, but most of the code remained the same between the different platforms. Preliminary results obtained from executing parallel programs proved the feasibility of `DSM-PEPE` as a parallel machine.

2 Design Issues

A software, page-based DSM system allows a collection of independent computers to share a single, paged, virtual address space. There are several issues that must be considered when designing such a DSM system. Some of them are particular to the underlying operating system, but most are platform independent.

In this paper, we show the design and implementation of a DSM system from this perspective. Modules are grouped in layers, in such a way that operating-system dependency is reduced. Our goal is to obtain a system that can be ported easily from one operating system to another, on top of the same hardware.

2.1 Pages and Consistency

Pages are usually cached on several nodes, in order to increase performance. This replication introduces a coherence problem that is managed by a consistency protocol enforcing a particular consistency model. The consistency model states the memory behavior in the presence of conflicting accesses across processors [4]. Relaxing the model allows better performance, under certain programming restrictions [5].

Consistency protocols must be kept separate from the page-management module. Otherwise, each time a protocol is included, it would be necessary to write page-management primitives for it. Some consistency protocols rely exclusively on the triggering of page-fault events, for example, sequential-consistency protocols. Protocols for relaxed consistency models usually take consistency actions when synchronization operations occur, for example, protocols for release and entry consistency.

Implementing a sequential-consistency protocol involves writing handlers for page-fault events on read and write operations, and for serving requests from a remote processor. For example, a write fault triggers a local event that is managed by the local handler. In one approach, the handler obtains an up-to-date version of the page, and instructs the other processors to invalidate any cached copy. Invalidation notices as well as the request for the page are handled by the remote handler. Read faults produce similar scenarios.

Sharing granularity affects system performance [6]. Multiwriter protocols reduce the impact of false sharing on performance when page granularity is used [7], but they are difficult to implement. The simplest approach is to allow false sharing and to use an invalidation protocol as described above.

Page management is accomplished through the virtual-memory interface of the operating system. The system must be capable of changing the status of virtual pages in order to make them valid or invalid, as well as read-only or writable. A small set of operating-system-independent primitives must be available to the upper levels of the system, in order to make high-level layers portable.

Exception handling allows the DSM system to catch access faults to pages that require consistency management. Application programs run guarded by an exception handler. Each time a page fault occurs, the DSM system evaluates the reasons and takes the necessary actions to resume the program.

2.2 Underlying Message Passing

In a multicomputer, processors must communicate through message passing. It is possible to use PVM or a user-level library based on MPI, to avoid the use of low-level socket communication. When using sockets, it is desirable to hide operating-system dependencies introducing a higher-level layer. Hence, code built on top of this new layer can be used across platforms.

DSM systems exchange messages of two types. First, consistency-related messages, controlled by the consistency protocol. Second, synchronization-related messages to implement distributed locks and barriers. All data could be directed through a single socket, but this may produce interference between the two types of messages. A more portable approach involves the use of an abstraction built on top of sockets: a post-office object, which uses an exclusive socket to communicate with its peers. A few primitives are available through the post office, allowing sending and receiving point-to-point messages, as well as broadcasting.

Communication through sockets can be connectionless or connection oriented. Connectionless sockets are efficient for scattered communication. DSM systems show high rates of message exchange. Hence, connection-oriented sockets seem to be more suitable. However, connection setup introduces unwanted overhead; thus, it is desirable to keep connections open.

2.3 Application View and Memory Regions

The DSM approach is more attractive than message passing to write applications for a multicomputer, since most programmers find shared memory easier to use.

To sustain this fact, applications must see the global memory space, as they see the shared memory in a multiprocessor. The use of memory pointers is an easily understood and portable interface. Programmers are familiar with the use of dynamically allocated memory. This approach is suitable for implementing the DSM system, since addresses stored in pointers can be easily mapped from the process space to the system global space.

It is convenient to group related variables in regions. A region is a high-level abstraction used to provide flexibility to the application programmer. Memory allocated from a single region is managed by the same consistency protocol. However, pages contained in different regions can be handled by different protocols. When the consistency protocol does not match the data-access pattern of an application, performance can be degraded. It is desirable to have a set of protocols with particular characteristics and be able to use them as needed. Sometimes, it may be necessary to use more than one protocol in the same application. This facility can be useful when the access pattern of an application changes during its execution. Hence, it is possible to match dynamically these data-access patterns. In general, the use of regions allows specifying extra information for portions of the address space. For instance, the entry consistency model requires that programmers associate each shared variable with a lock [8]. This association can be done by grouping together, in the same region, all related variables.

2.4 Multithreading and Migration

Multithreading in DSM systems has been widely studied [9,10,11]. Application-level multithreading reduces remote latency in DSM systems, by overlapping communication and computation [12]. Besides, multithreading helps to enhance program structure.

Threads can exist at kernel level or at user level. User-level threads are portable, flexible, and can be implemented efficiently. Only the thread-switching facility is actually dependent on the underlying hardware and operating-system platforms. In a DSM system there is a single address space. Also, there must be a single global thread space, and threads must be unique across processors. Synchronization should coordinate threads no matter where they are running.

Thread migration is another topic widely studied. In general, process migration allows load balancing and dynamic reconfiguration. In a DSM system, thread migration enhances data locality, by moving threads to the data they need [9,10]. This locality reduces consistency-related communication. However, it is necessary to implement an effective migration policy. In a DSM system, the primary goal is not load balancing, but to enhance data locality. In order to do so, data-access patterns of applications must be accounted for dynamically. Some DSM systems allow multithreading and thread migration at the application level, for example, Millipede [13,10] and DSM-Threads [11].

Implementing thread migration requires considering several issues. We focus our study on computers having the same hardware architecture, running the same operating system. Hence, some problems do not arise, for example, data and address representation. Data-addresses correspondence is not a problem since shared data is stored in the global shared-memory space. Code-addresses correspondence is easily handled, by having all threads start at all processors, even if they are actually running in only one of them. Each running thread has a suspended peer in each of the other processors, waiting to receive it in the event of a migration. Stack migration can be easily accomplished, since threads are implemented at the user level. The threads library provides functionality to suspend a thread and recover its stack, as well as to resume a suspended thread with an updated stack.

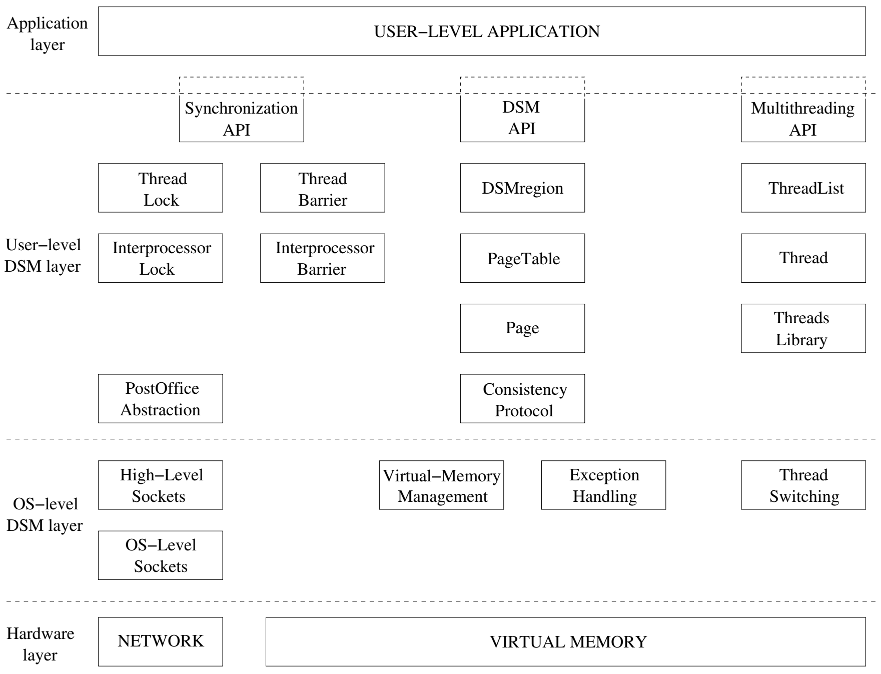

Fig. 1. DSM system architecture

2.5 Synchronization

Synchronization is a key issue when programming a system with shared-memory semantics. At least two kinds of synchronization primitives are required: locks, to provide mutual exclusion, and barriers, to control interprocess progress.

In a multithreaded DSM environment, there are two levels of synchronization. At the bottom level, interprocessor synchronization allows processors to coordinate with each other. At the top level, and visible to the application program, thread synchronization allows global threads from the application program to coordinate with each other.

Thread synchronization can be built on top of interprocessor synchronization. For example, thread barriers must block arriving threads until all threads from all processors have arrived to the barrier. One possible implementation provides local barriers within each processor. When all local threads have arrived, the processor notifies a global interprocessor barrier about the arrival. The global barrier blocks the processor until all processors have notified their arrival. A release notice from the global barrier triggers the release of all blocked local threads in each processor. Locks can be implemented in a similar way, using a global lock and local locks within each processor.

2.6 Layered Architecture

Fig. 1 shows the suggested layered architecture for a portable DSM system. Interaction is accomplished through three *Application Programming Interfaces* (API). The synchronization API provides lock and barrier synchronization to global threads. These primitives can be implemented on top of the interprocessor synchronization functionality. The DSM API allows applications to create shared-memory regions and to allocate memory from them. Lower-layer modules implement page functionality and coherence. The consistency protocol is kept apart from all other modules, making it independent, easily replaceable, and allowing the coexistence of different protocols on the same user application. The multithreading API is implemented on top of a global threads layer, which, in turn, is implemented on top of a user-level threads library. Only the thread-switching component is operating-system dependent.

Being a distributed system, most of the functionality relies on the communication modules. The post-office abstraction hides communication details from upper-level layers. Portions of the socket layers need to be rewritten for each operating system.

3 Implementation Issues

Most of the DSM-system components are operating-system independent. A few modules, namely, page management, exception handling, message passing, and thread switching, are dependent in some degree on the underlying platform. Our system is portable in the sense that only certain portions of these components are different between the ports for Linux and Windows.

3.1 Portable Modules

Page management is the core of the DSM system. Application programs use the shared memory through regions. Consistency of a region is managed by a consistency protocol, specified when the region is created. The size of the region defines the number of pages it contains. The page object links a memory page to its region and to the consistency protocol. Consistency for a page is ensured by a specialized thread. Hence, message-handling threads are not blocked unnecessarily. The DSM API includes functions to create regions, specifying its size and consistency protocol, and to allocate memory from a region, specifying the amount of memory needed.

Consistency protocols are implemented taking the page class as their basis. A new subclass must be derived from the page class, and handlers for the consistency events must be coded. Any additional data that the protocol may need is defined within the subclass. For example, a sequential-consistency protocol may require a hint of the probable owner of the page; this information would be stored in the page subclass.

Interprocessor barriers use a centralized coordinator. The coordinator receives messages from all processors arriving at the barrier, and broadcasts a release notice once it has received all of them. The implementation of interprocessor locks is distributed. Each node keeps a hint of the probable holder of the lock, and sends a request when it wants to acquire the lock. The lock holder queues the requests and delivers the queue when releasing the lock. Once the lock is released and sent to another processor, the former holder forwards any request it receives to the node to which it relinquished the lock.

Barriers at the thread level are implemented on top of interprocessor barriers. Thread locks are implemented using local queues and *alien threads*. An alien thread stands for a remote thread requesting the lock from another processor. The alien thread waits at the queue of the holder processor; once it gets the lock, it sends the lock to the remote thread it represents. When the lock migrates, the queue is transferred with it.

Threads are global entities, implemented on top of a user-level library [14]. A pool of threads is created in every processor when the application program starts running. All threads start execution and block until they are needed. When the program forks a thread to execute some particular function, the thread is setup in all of the processors atomically, although it is run only at the location that did the fork. The other instances stay suspended, waiting for the thread to migrate in. Each thread is given a unique system-wide identifier, which can be used to join with other threads. Thread migration is supported through the suspended state; only suspended threads can migrate. The migration facility extracts the thread state, including its stack, and sends it to the target processor. Once there, the thread is resumed in the same state in which it was when it was suspended.

Communication is done exclusively using the post-office abstraction. Two global post offices exist in each system node. The first is used exclusively for consistency-related messages. The second is used for the exchange of synchronization-related messages. Reception of messages in each post office is handled by an independent thread.

3.2 Non-portable Modules

Page management and exception handling are highly dependent on the operating system underneath. For page management, there are a few primitives that can be abstracted to implement the DSM system on top of them. Basically, we need to be able to set pages as invalid, as well as to set page protection to read-only or read-write. In Linux this page manipulation is possible using the `mprotect` system call. Virtual-address correspondence is accomplished manipulating the heap. In Windows, we must use some functions from the virtual-memory interface: `VirtualAlloc`, `VirtualQuery` and `VirtualProtect`. Virtual-address correspondence is guaranteed using memory-mapped files.

Exception handling is required to catch faulting accesses to memory, in order to handle those faults related to the DSM management. Application programs run protected by an exception handler responsible for interrupt interception. Windows provides a structured mechanism which is suitable for this purpose. It is possible to protect any code, in particular application code, with a handler for virtual memory exceptions. In Linux there is no such a structured way of handling exceptions, but we can use the Unix signal-handling mechanism instead. In particular, it is possible to install a handler for the `SIGSEGV` signal, raised on faulting memory accesses; the `signal` system call fulfills this purpose.

Communication is not isolated from portability issues. Although socket communication is relatively high level, Windows implementation of sockets differs from Linux and Unix implementations. These differences must be hidden in a software layer that provides a portable interface to the upper layers.

Finally, thread switching is the only component of the thread library that is not portable across operating systems. A few lines of assembly code accomplish this task, and had to be written for each operating system.

4 Preliminary Results

`DSM-PEPE` runs on top of two of the most popular operating systems nowadays. We succeeded in implementing a system that works on both platforms with minor changes. This section presents results from executing a simple parallel application on both systems, and a comparison of performance between them. Also, we show speedups relative to a sequential, equivalent program.

The testbed is a set of up to 16 personal computers with the same configuration: Intel Pentium 4 running at 1.66 GHz, 256 MB RAM, and 512 KB L2 cache; connections are switched at 100 Mbps. We used both Windows 2000 and RedHat Linux 8.0.

The application chosen is a simple integer square-matrix multiplication. Two different matrix sizes were used: 1024×1024 and 2048×2048. No special optimization was done. Computation is distributed splitting the result matrix in slices of the same size, using a striped approach. Each processor is responsible for computing the values in the slice assigned to it. This scheme produces a highly parallel task; source matrices are accessed only for reading, and there are no race conditions, since the writing of the result is distributed among the processors. The turnaround time of the program was measured using 2, 4, 8, and 16 processors, for both matrix sizes. Also, a sequential version of the program was run in a single processor, in order to calculate speedups for the parallel versions. Measurements were taken five times to produce an average value; variance was negligible.

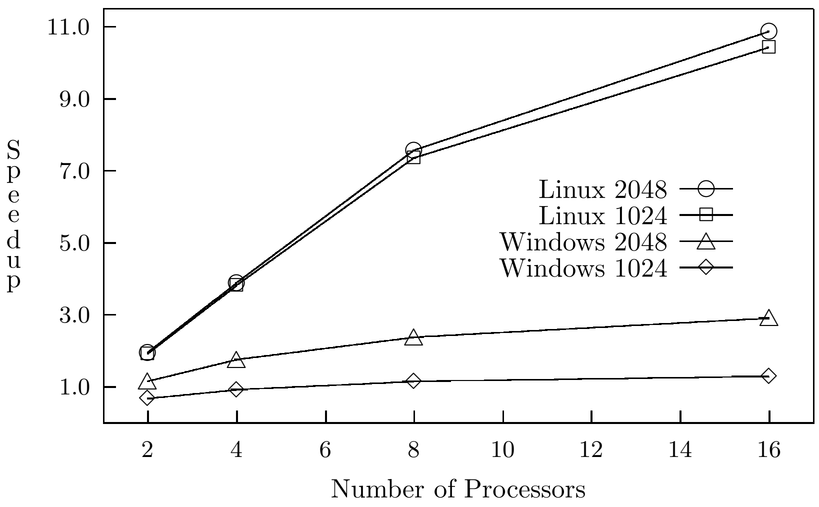

Fig. 2. Speedup for two problem sizes running on Linux and Windows

Fig. 2 shows a comparison for the 16 experiments performed. Speedups are clearly higher for the application running on the Linux version of the system, producing a high degree of efficiency. Since most of the code is identical across versions, as well as is the hardware platform, the difference is due to operating-system-dependent components of the system. Preliminary evaluation shows that socket communication is the major responsible for the lower performance on Windows. Moreover, exception handling and virtual memory management mechanisms are at a higher level in Windows than in Linux, producing additional overhead in the Windows version of the system.

5 Concluding Remarks

Portability is a key issue when designing and implementing a software DSM system. The layered design of `DSM-PEPE` allowed us to reuse most of the code for both the Linux and Windows versions of the system. Only those components responsible for low-level socket communication, exception handling, virtual-memory management, and thread switching needed to be rewritten across platforms.

System maintenance is easy, because of the modular design. A specially designed interface for the consistency-related events, allows to add new consistency protocols without affecting other parts of the system.

Our preliminary results confirm the potential of a software DSM system as a parallel computing environment. Programming for the system is easy, and the cost of building the virtual machine is low. In the future, we expect to perform an extensive set of experiments using different applications extracted from known benchmark suites.

Slightly heterogeneous distributed systems can be built using `DSM-PEPE`, that is, systems composed of computers running different operating systems on top of the same hardware. Hardware homogeneity allows an easy exchange of data between machines running different operating systems. An open area of study involves thread migration in such a slightly heterogeneous system.

Thread migration presents several problems in the context of relaxed consistency protocols. Currently, we are studying the relation between relaxed memory protocols, thread migration, and the data-access patterns of the programs.

Acknowledgements

The first author acknowledges the support of Fondecyt, under Grant 2990074, and of Universidad de Talca.

References

1. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems **7** (1989) 321–359
2. Lu, H., Dwarkadas, S., Cox, A.L., Zwaenepoel, W.: Quantifying the performance differences between PVM and TreadMarks. Journal of Parallel and Distributed Computing **43** (1997) 65–78
3. Lo, V.: Operating systems enhancements for distributed shared memory. Advances in Computers **39** (1994) 191–237
4. Adve, S., Gharachorloo, K.: Shared memory consistency models: A tutorial. Technical Report ECE-9512, Rice University, Houston, TX (USA) (1995)
5. Adve, S., Hill, M.: Weak ordering: A new definition. In: 17th Annual International Symposium on Computer Architecture, ACM. (1990) 2–14
6. Torrellas, J., Lam, M.S., Hennessy, J.L.: False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers **43** (1994) 651–663
7. Keleher, P.J.: The relative importance of concurrent writers and weak consistency models. In: 16th International Conference on Distributed Computing Systems. (1996) 91–98
8. Bershad, B.N., Zekauskas, M.J.: Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA (USA) (1991)
9. Thitikamol, K., Keleher, P.: Thread migration and communication minimization in DSM systems (invited paper). Proceedings of the IEEE **87** (1999) 487–497
10. Itzkovitz, A., Schuster, A., Shalev, L.: Thread migration and its applications in distributed shared memory systems. Journal of Systems and Software **42** (1998) 71–87
11. Mueller, F.: Distributed shared-memory threads: DSM-Threads. In: Workshop on Run-Time Systems for Parallel Programming. (1997) 31–40
12. Thitikamol, K., Keleher, P.: Per-node multithreading and remote latency. IEEE Transactions on Computers **47** (1998) 414–426
13. Friedman, R., Goldin, M., Itzkovitz, A., Schuster, A.: Millipede: Easy parallel programming in available distributed environments. Software: Practice and Experience **27** (1997) 929–965
14. Cormack, G.V.: A micro-kernel for concurrency in C. Software—Practice & Experience **18** (1988) 485–491