Proceedings of International Conference on Distributed Systems, Software Engineering and Database Systems

Multi-threaded Design for a Distributed Shared Memory System

Ce-Kuen Shieh, Jyh-Chang Ueng, Su-Cheong Mac, An-Chow Lai, Tyng-Yeu Liang
Department of Electrical Engineering,
National Cheng Kung University,
Tainan, Taiwan

Abstract

This paper describes the design and implementation of a multi-threaded Distributed Shared Memory (DSM) system, called Cohesion, which provides high programming flexibility and latency masking, and supports load redistribution. Cohesion differs from many other DSM systems by providing a virtually shared address space, instead of a shared variable model, in which user and system data can be located. This address space is partitioned into regions, each maintained by a distinct memory coherent protocol. User and system data with similar behaviors can be placed in the same regions and maintained by a protocols that suits their nature, thereby improving the system performance. The design issues such as synchronization and thread management have been reconsidered to support multi-threading. The results of Successive-Over-Relaxation and Quick Sort on Cohesion show that multi-threading has better performance than single-threading since communication can be overlapped with computation.

1. Introduction

With continue advance on Distributed Shared Memory (DSM) [1] technology and the fast approach of high speed network, DSM systems become an appealing alternative for parallel processing. The high availability and parallel programming model similar to those provided by multiprocessor systems make DSM very popular. Nevertheless, the research of past DSM systems [2][3] are focused on the proposal and realization of new consistency models and consistency protocols. Their user interfaces are kept as simple as possible in order to ease implementation. Particularly, most of these systems allow only one user thread to be executed on a processor, i.e., the total number of threads that can be created in an application is determined by the number of processors. We categorize this kind of DSM systems as single-threaded DSM systems. In fact, a multi-threaded DSM system that allows users to create more than one threads on a processor has two advantages over a single-threaded one: programming flexibility [12] and latency masking [10]. In this paper, a multi-threaded DSM system, named Cohesion, is proposed. To further improve system performance, dynamic load redistribution by thread migration is also supported in our system.

There are several existing and developing DSM systems that also support multi-threading [10][11][12]. Our system resembles these DSM systems in latency masking by overlapping communication and computation. On the other hand, our system differs from most multi-threaded DSM systems by providing a shared address space, instead of a shared variable model, for users. User data can be statically or dynamically allocated from this address space. No construct or special primitive is required to declare that a variable is shared. In addition, we locate the system data, e.g., thread control block (TCB) and stack, in this virtually shared address space instead of the private memory. This leads to another advantage when thread migration is considered: offset conversion is avoided because all nodes share the same address space, and the system data are implicitly moved by the memory manager to the destination node during migration time.

Putting user and system data into a shared address space maintained by a single coherence protocol may result in performance loss, since different user and system data possess different behaviors, i.e., different access patterns. To solve this problem, our multi-threaded DSM system employs multiple consistency models. Programmers and system designer can freely choose a protocol that suits their data. Moreover, affinity scheduling, hierarchical barrier, two-level memory allocation, and a modified delayed-update queue handling are employed to further improve system performance. Our thread scheduler allows the overlapping of communication and computation by switching to another ready thread when the current one is blocked waiting for the response. Several programs were employed to verify the effectiveness of our design, and the results show that system performance is indeed improved by overlapping communication and computation.

The remaining sections of this paper are organized as

This work was supported by the NATIONAL SCIENCE COUNCIL, project number 84-2213-E-006-011.

follows. Section 2 describes the system overview of Cohesion. We discuss the related issues and our decisions in designing our thread system in section 3. Our implementation follows in section 4. Section 5 shows the performance result of our multi-threaded DSM system in the experiments. We conclude in section 6.

2. System overview

Cohesion is a DSM system supporting two memory consistency models, namely, eager release consistency [2] and sequential consistency models [1]. It provides a virtually shared address space that is accessible by any nodes. In addition, Cohesion consists of an object-oriented runtime thread system, providing a parallel programming environment in the user space. In the following subsections, the shared address space and the user interface will be described.

2.1 Shared address space

Cohesion provides a shared memory address space for programmers, while some DSM systems [2][12] allows users to explicitly declare which variables are shared. In these DSM systems, a special primitive or construct is provided for programmers to declare that an object or a variable is shared by all nodes. This is unnecessary in Cohesion, in which every object is naturally shared by all nodes. With this shared address space model, our memory allocation interface is very close to those in shared memory multiprocessors. This greatly simplifies programmers' work in porting or writing parallel programs. User and system objects can be statically and dynamically allocated in this address space.

To support multi-threading, the shared address space is divided into three regions, i.e., release, conventional, and object-based memory, as shown in figure 1. The release memory is provided to relieve the performance degradation caused by false sharing of user's data. A delayed-update protocol [2] is employed with a page granularity to maintain consistency in this region. The conventional memory employs Kai Li's write-invalidation protocol [1] to maintain sequential consistency. Some system data of our thread system are stored in this regions to simplify the implementation of thread redistribution. From the users' point of view, it is much easier to port an existing shared memory parallel program to a DSM system with a conventional memory region, because the usage of this memory is close to that of physically shared memory. However, memory accesses in this memory region may possess higher overhead than that in release memory region, due to the sequential consistency maintenance. If the performance of a newly ported program is unsatisfactory, programmers can easily relocate their data from the conventional memory region to the release memory region. The object-based memory is managed with a migratory protocol. This memory region is not directly accessible by users. It is dedicated to other system data objects in our thread system, such as TCBs, which makes thread redistribution easier because objects in this region are maintained by a migratory protocol and there is no false sharing of objects.

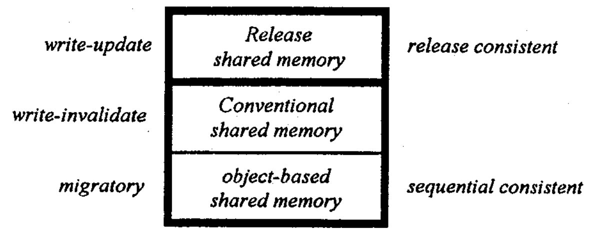

Figure 1. Shared address space in Cohesion

2.2 User interface

The user-interface of our thread system is similar to PRESTO [5], built at the University of Washington. It consists of the usual thread primitives such as fork, join, lock, barrier, dynamic memory allocation, etc. These primitives are nearly the same with those provided by shared memory multiprocessor systems. In order to present a clearer illustration, the entire flow of execution in a Cohesion application is shown in figure 2.

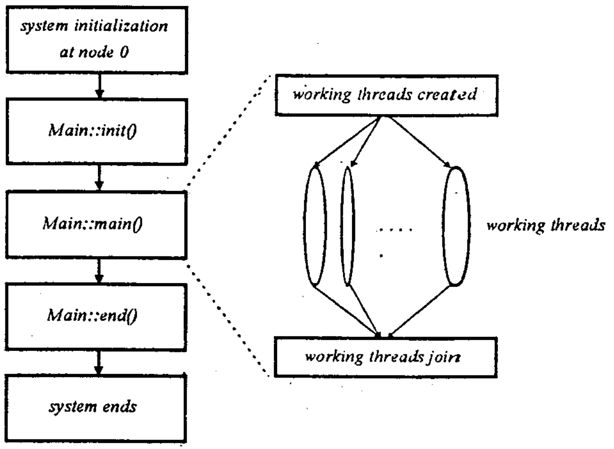

Figure 2. Execution flow of a user program

Typically, users may dynamically allocate shared objects in the function *Main::init()*, and locate the main thread in the function *Main::main()* where working threads are forked and joined. Both pre-forking model and dynamic forking model are supported. In the pre-forking model, the working threads are forked by the main thread. In the dynamic forking model, a working thread can dynamically fork other working threads. Lastly, there is the optional function *Main::end()*, allowing users to make a concluding run of their applications after all working threads have joined.

Memory annotation

Since there are two user-accessible memory space in Cohesion, there must be a way for users to distinguishably allocate different object types. This is achieved by data annotation. Every shared object in Cohesion is considered as a data item in a class. If a object inherits the **Release** class provided by the system, the memory consistency model applied to the object is set to *release*. A shared object that has not inherited any system class is kept sequentially consistent by default. In this way, compiler modification or special preprocessor is not needed. It is noted that the type of a shared object may not be changed once it is set. This kind of annotation is convenient since inheritance is a feature of object-oriented programming, and it allows a programmer to relocate data from the conventional memory region to the release memory region simply by inheriting the **Release** class in his data objects. To briefly describe how our data annotation work, an example is given in figure 3.

```
// declaring the Matrix as a release object
class Matrix : public Release {
      private : shared data items;
      public : member functions;
};
// declaring the Sor as a sequential object
class Sor {
      // this is default without inheriting
      private : shared data items;
      public : member functions;
};
```

Figure 3. An example of data annotation

3. Design methodology

To design an efficient multi-threaded DSM system, issues such as thread management, dynamic memory allocation, and synchronization should be carefully considered. The following covers these issues in detail.

3.1 Thread management

In Cohesion, thread management can be divided into three main parts: thread creation model, thread scheduling, and thread migration. Thread creation model defines how threads are created in a program. Thread scheduling handles load distribution and context switching. Thread migration is the basic mechanism for load redistribution.

3.1.1 Thread creation models

In our system, threads can be created statically or dynamically. In the static (pre-forking) creation model, programmers fork the required threads before the computation starts; while in the dynamic creation model, threads can be dynamically forked during computation. Static creation model is suitable for applications whose workload can be statically determined, e.g., Successive Over Relaxation (SOR). Dynamic creation model is well suited for problems with either static workload or dynamic workload, e.g., divide-and-conquer problems. Programmers can freely choose amongst these models or even employ both simultaneously in their programs.

3.1.2 Thread scheduling strategies

We adopted the idea of affinity scheduling (AFS) [6] in Cohesion because this algorithm can be modified to suit most of the subtle differences in circumstances arising in a distributed environment. Each processor in our system possesses a local scheduler and a local ready queue. Initially, some working threads are statically created in *Main::main()* before the computation starts. These threads are assigned to the local queue of each processor. The number of threads assigned to each processor is equal to the number of the total threads divided by the number of processors in the system, so that the workload is divided among the processors as evenly as possible, i.e., if there are M user threads and N nodes, every node will get M/N threads. These threads can be distributed in batch mode or interleave mode. In batch mode, the threads are distributed so that the first M/N threads are located at the first node, the second M/N threads at the second node, and so on. In interleave mode, the threads are assigned to the nodes in a round-robin fashion. On the other hand, a dynamically created thread is located at the ready queue of the creator node. However, there will probably be the case that the load distribution in the system differs from time to time due to the characteristics of some parallel algorithms or difference in processor speed. As a result, some processors may be idle waiting for the synchronization barrier or a response of shared memory misses. This problem may probably be resolved by adding a load balancing function. We have proposed and experimentally evaluated a load balancing method, called Dependence-Driven Load Balancing (DDLB), based on thread migration for Cohesion. The details in DDLB are covered in [14]

3.1.3 Thread migration supporting

When a thread is migrated, the user data, the TCB, and the stack associated with the thread should be moved to the destination node. User data are stored in the virtually shared address space, so the migration of this portion can be implicitly done by Cohesion. On the other hand, the TCB and the stack of a user thread can be stored in private or shared memory. If they are stored in private memory, the thread system has to explicitly move them to the target node via network when thread migration occurs. Moreover, the offset of these data structures in memory may be different between the two nodes participated in the thread migration, and offset conversion is necessary, which increases system complexity. In contrast, if these system data are put in the global address space, thread migration is simple because every node can directly access system data. In Cohesion, these two data structures are placed on shared memory: the TCB of a user thread is placed in object-based memory, while its stack is located in conventional memory. The main reason that the TCBs are located in the object-based memory is to avoid memory offset conversion, because an object address always points to the same object in any node. Putting the TCBs in conventional memory can also avoid offset conversion, but false sharing may arise because a TCB is much smaller than a page. On the other hand, we place the stack of each user thread in conventional memory to avoid overhead in object management. Every stack is page-aligned, so that false sharing can be avoided.

3.1.4 Overlapping of communication and computation

The thread scheduler masks synchronization latency by overlapping computation and communication. Synchronization in a DSM system may not be satisfied immediately if the requested lock is held by another node or the requested barrier is not yet completed. In addition, when a lock is released by a thread, the update of the release memory pages produced by the releasing thread are flushed to other nodes. As a result, synchronization always induces network message exchange and long delay. In our latency masking technique, when the scheduler notices that the current executing thread invokes a synchronization event that cannot be satisfied immediately, another thread from the ready queue is executed and the current one is blocked.

Overlapping communication and computation may result in performance gain. However, this gain does not come with no cost. There is the overhead of thread creation, context switching, and increased amount of synchronization [10]. Whether the gain can surpass the overhead is determined by the program's behaviors and the thread's granularity. In section 5, we will show two examples in which overlapping of communication and computation can indeed improve applications' performance.

3.2 Dynamic memory allocation consideration

As in most applications, data are usually dynamically allocated and assigned after the programs starts. Since a DSM system is designed to provide a shared memory image via message passing, more care should be devoted to prevent from unnecessary network traffic that leads to performance loss.

Dynamic memory allocation can be managed in a single-level or two-level mechanism. Assuming that we employ a single-level mechanism, all dynamic memory allocation are handled by a centralized memory manager. Every memory allocation induces network messages, resulting in certain overhead. In a multi-threaded DSM system, this overhead is even heavier since there are much more threads in the system. To relieve this problem in Cohesion, we adopts a two-level memory allocation mechanism, which applies for the dynamic allocation for all types of memory. When a memory allocation is first invoked, the local thread system allocate a large chunk of shared memory from the central manager. This memory chunk is then maintained by the local thread system and the requested amount of memory is returned to the requesting thread. All subsequent memory allocation requests are then fulfilled locally from this memory chunk. When the local memory chunk is used up, another chunk is fetched by the thread system.

3.3 Synchronization

Synchronization is a critical component in a thread system that affects performance, because it usually induces network message exchanges and idle waiting. We have provided several synchronization mechanism, such as lock, barrier, monitor, etc. In this paper, we focus on locks and barriers, which are two frequently used synchronization primitives in parallel applications.

Lock

In our thread system, locks utilize a queue-based algorithm [7], in which every lock has an owner. The owner of a lock is the processor that acquires it. When a thread acquires a lock, a message is sent to its owner and the thread is blocked. Later when another thread at the same node acquires the same lock, instead of sending another request to the owner, the thread simply waits for the lock locally by a proxy. This mechanism will reduce a large amount of network messages in a multi-threaded environment, since only one message per node is needed instead of one message per thread. In order to relieve from bottleneck and to scale well, the ownership of each lock is maintained in a distributed scheme [7].

Hierarchical barrier

Traditional barriers [7] may have high overhead in a multi-threaded DSM system. Let us consider the case that there are more than one threads on each node wishing to synchronize at a traditional barrier. Each thread send a message to a central manager notifying its arrival at the barrier. The amount of network messages generated is directly proportional to the number of threads. To deal with this problem, we have developed a hierarchical mechanism for barrier, named as hierarchical barrier. In this mechanism, a thread is suspended when it arrives at a barrier. When all relevant threads in a node arrives at the barrier, an arrival message is sent to a central manager. When the central manager has received the arrival messages from all nodes, a barrier-completed-notification is broadcasted. In this way, the number of message is directly proportional to the number of nodes.

4. Implementation

In the following sub-sections, the implementation of related components of our thread system are shown.

4.1 Cohesion components

Besides the thread system, there are other four main components [4] in Cohesion, i.e., the reliable communication subsystem, the upcall supporting subsystem, the memory coherence manager, and the object server, as shown in figure 4. The reliable communication subsystem provides a reliable and efficient vessel to exchange messages among nodes. Upcall supporting subsystem provides services similar to the SIGNAL in UNIX, notifying the user-level event

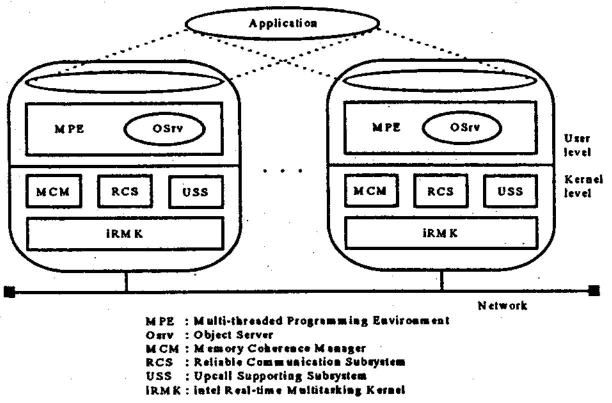
Application
MPE
OSrv
MCM
RCS
USS
iRMK
User level
Kernel level
Network
MPE : Multi-threaded Programming Environment
Osrv : Object Server
MCM : Memory Coherence Manager
RCS : Reliable Communication Subsystem
USS : Upcall Supporting Subsystem
iRMK : intel Real-time Multitasking Kernel

Figure 4. System Overview of Cohesion

handlers of the kernel events. The memory coherence manager enforces the consistency of the shared virtual memory space with two handlers, i.e., the conventional and the release handlers. The conventional handler deals with page faults and consistency maintenance of the conventional memory. The release handler processes page faults in release memory by just notifying the object server, which is actually responsible for maintaining consistency of the release memory, via upcall. In addition, the object server also manages objects in the object-based memory in a way similar to Amber [8]. All these components are based on a commercial real-time kernel, iRMK[9]. In Cohesion, there are both kernel-level and user-level threads. The kernel threads are only for system use and are managed by iRMK. In practice, user threads created by a program are run in user space and scheduled onto a kernel thread that serves as virtual processor.

A delayed-update queue per thread

Similar to other DSM systems supporting a release consistency model [2][3], Cohesion employs a delayed-update protocol [3] to handle the modification of release memory pages. Every thread in Cohesion has its own delayed update queue, instead of sharing a single queue by all threads in the same node. This model is more efficient and easier to implement than the sharing model. If all threads in a node share a single delayed update queue, the queue is traversed to find out which pages have been modified by the thread that executes the *release* operation. The queue may become very long and the searching is then time-consuming. When every thread has its own queue, the update of every page recorded in the queue is flushed when the thread executes a *release* operation. Searching is not required.

Page fault handling

To illustrate how page fault is handled, Cohesion's page fault handling and upcalling mechanism is shown in figure 5. Since the algorithm for the conventional handler is so familiar [1], the following description is concentrated on the release handler and the relation between these two handlers.

When there is a read fault in release memory, the release handler fetches the page and set its access right as *read-only*. Since a read miss does not affect buffering or write sharing in the protocol, no upcall is initiated and the control is returned. When a write fault occurs, the release memory handler acquires the page if it is not present. The handler then upcalls to the object server together with the requested page number. The upcalling handler in the object server may realize that the page is dirty and the

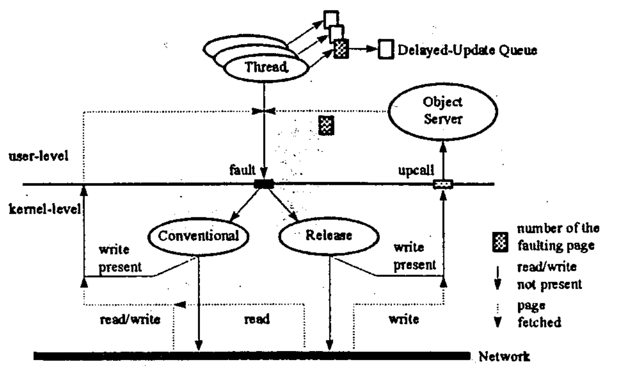

Figure 5. Page fault handling and upcalling

page number is appending to the faulting thread's delayed-update queue. Subsequently, the control will be returned to the faulting thread.

4.2 Thread migration mechanism

Thread migration is done simply by moving the TCB and the stack of the thread to the target node. As mentioned in section 3.1, the TCB and the stack of a thread are located in object and conventional memory space, respectively. Each TCB is an object and its consistency is maintained by the object server. Stacks are page-aligned and their size are fixed at one memory page, which is 4Kbytes in intel x86 processors, to avoid false sharing. From our experience, the stack of a user thread seldom exceeds a memory page, and it can be easily expanded to multiple pages. When a thread is migrated, its TCB is explicitly sent to the target node. A pointer to this TCB is then added to the ready queue of the target node. The thread will eventually be resumed and when it accesses its stack, a page fault is generated because the stack has not moved with the thread. The conventional memory manager will then fetch the page containing the stack. By putting the stacks in conventional memory, stack migration can be easily done and unnecessary stack movement can be reduced. For example, thread 1 is migrated from node A to node B, and is then migrated to node C before resuming execution in node B. If the stack is placed in conventional memory, it will be moved directly from node A to node C because the stack is moved only when it is accessed after resumption. If the stack is moved with its thread during thread migration, it will be moved twice. However, putting a stack in shared memory may induce extra overhead. The content of a stack may not fully occupy a page. With the above implementation, the null data in the stack's page are also moved by the memory manager during stack migration. The effectiveness of our mechanism is determined by the program's behavior, e.g., how frequently are the threads migrated and the usage of the stack. Nonetheless, our approach is very simple in implementation.

4.3 Dynamic memory allocation

To dynamically allocate shared memory in Cohesion, users employ the overloaded *new()* which will invoke a different routine according to the annotated variable type. However, after doing relevant book-keeping for the annotated variable, *malloc()* in the thread system is called. *malloc()* will check if there is enough free space which is preallocated. If not enough free space is available or this is the first allocation request, a trap to the kernel will be generated via *sbrk()* system call with a parameter *type* to ask for a chunk of the shared memory from a centralized manager. Subsequently, free page frames will be mapped into the shared address space to allow the expansion of data segment controlled by *sbrk()*. Typically, the parameter *type* provides a key to the *sbrk()* call such that it will set the conventional and release pages as *read-only*, and the object-based pages as *read/write*. The routines following *sbrk()* are different for different memory types. For a conventional memory allocation, the requested amount of memory is returned to the user application. While for the release and object-based memory, further management such as appending a header before the requested memory are desired.

5. Performance evaluation

To verify the effectiveness of our multi-threaded DSM system with overlapping of communication and computation, two experiments have been carried out with two scientific computing algorithms, namely Successive Over Relaxation (SOR) and quick sort (QSort). The programs are executed on 4 90MHz Pentium-based microcomputer, connected by 10Mbps Ethernet. The network is isolated from the campus network throughout the experiments to avoid interference. In our environment, a context switch takes 9 microseconds and a page fault takes 15 milliseconds to process.

For each of the two applications, we have written a single-thread-per-node (1T), a 2-threads-per-node (2T), and a 4-threads-per-node (4T) version. The effect of latency masking in our thread system can be shown by comparing these three versions. In all testing, there was a main thread at processor 0. Working threads were forked by this thread. The main thread then waited until all threads finished their jobs. The start time was recorded when the main thread was started, and the end time was recorded when all threads had rejoined. The elapsed computation time is the difference of these two times.

SOR

In SOR, there are two matrices in the program, which are in turn considered as current and scratch arrays before an iteration is started. Every element calculated for next iteration is written to the scratch array. In our experiment, there are 512x8192 floating point numbers in the pending area and there are twenty iterations presumed for convergence in the program. We choose this problem size to ensure that the idle time is long enough for the purpose of this experiment. In this problem size, the shortest idle time in the case 1T among 4 nodes is 2.4 seconds. In SOR algorithm, more data sharing at the boundaries implies more data are required to be exchanged among the nodes during execution. For example, in case 2048x2048, 1024x4096, and 512x8192 with 4 nodes, the total amount of network traffic are 1213500, 2188228, and 4124052 bytes, respectively. Therefore, more data sharing implies much time is required to flush data.

Figure 6 shows the elapsed computation time of SOR. Obviously, the performance gain by masking latency is very limited. This is because the idle time in SOR is relatively short compared to the computation time. However, it clearly shows that the most portion of idle time are masked in the case 4T.

QSort

In our experiment, there are 1048576 (1M) integers. The elapsed computation time is shown in figure 7. The improvement gained by masking latency is quite impressive in this application. This is because there are much more data that should be flushed by a thread at the end of a synchronization than that in SOR, which is illustrated in figure 8. As a result, the idle time is much longer in this case. For example, the shortest idle time in case 1T among 4 nodes is 12.75 seconds.

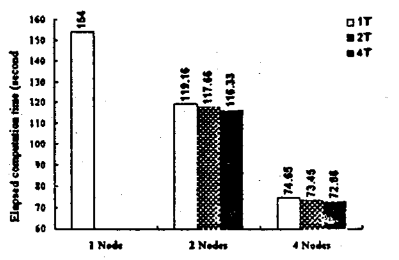

Figure 6. Elapsed computation time of SOR

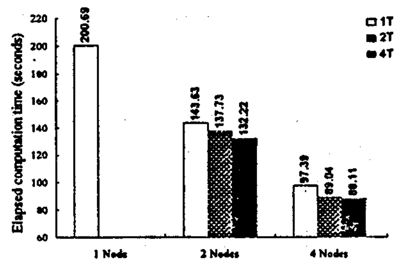

Figure 7. Elapsed computation time of QSort

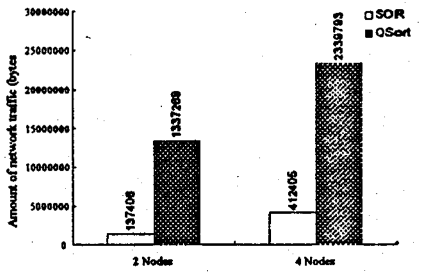

Figure 8. Amount of network traffic

6. Related work

Munin [3] is a DSM system that provides multiple consistency protocols. It differs from our DSM system by using a single-threaded programming environment. Although Munin can create several system threads in a node to handle messages, users are limited to create only one user thread at each node.

Quarks [11] is a user-level multi-threaded DSM system supporting multiple coherency protocols. The main difference between Quarks and Cohesion is that Quarks does not support thread migration. As a result, the locations of the system data is not critical since these data are not moved among nodes. Quarks employs a single-level memory allocation mechanism, while Cohesion utilizes a two-level memory allocation mechanism. Barriers in Quarks are managed in a centralized way, while Cohesion employs hierarchical barrier to reduce network messages.

Distributed Filaments [10] is another multi-threaded DSM system. A filament is a very lightweight thread that has no stack. Basically, there are two main differences between Distributed Filaments and Cohesion in multi-threading: (1) Distributed Filaments does not support release consistency, therefore its memory consistency management is independent of multi-threading. Cohesion provides a release memory space and its consistency maintenance should be accordingly modified to suit for multi-threading. (2) Distributed Filaments' programming model differs from those provided by shared memory multiprocessor systems. Users have to learn a new one and to rewrite existing parallel programs. Cohesion's programming model is similar to shared memory multiprocessors. Programmers can use the traditional interface in programming and the existing parallel applications can be ported easily.

Millipede [12] is a multi-threaded DSM system that provides multiple consistency models and dynamic load sharing. The two main differences between Millipede and Cohesion are the user interface and the location of system data. Millipede supports the ParC language, and a pre-processor is required to convert ParC to C. Millipede places system data in the private memory. The stack contents and the register values are transferred at migrated time. Offset conversion is avoided by ensuring that a stack will occupy the same address on all hosts. In Cohesion, the user interface is similar to PRESTO, and pre-processor is not necessary. System data are located in shared memory, so that offset conversion or memory space reservation is not needed. Only the TCB of the migrating thread is transferred at migrated time. After the migrating thread resumes execution, its stack is transferred by memory manager when it is accessed. In this way, unnecessary movement of stacks can be avoided. However, since the stacks in Cohesion are page-aligned and implicitly transferred by the conventional memory manager, the page containing the stack is moved when it is accessed after resumption, resulting in a slightly higher overhead compared to Millipede.

7. Conclusions

In this paper, we have demonstrated how to build a multi-threaded DSM system that provides high programming flexibility, latency masking, and supports load redistribution. With the multiple consistency models provided, we can neatly maintain the consistency of the system data of thread system such as TCBs and stacks according to their nature. Our programming model and user interface are similar to those in shared memory multiprocessors. The performance results show the improvement gained with latency masking technique, especially for applications with heavy network traffic such as QSort. Thread migration, which is the basic mechanism for load redistribution, is also provided. The combined effects of these features show that a multi-threaded DSM system has its advantages. A completely user-level multi-threaded DSM system is being developed in our laboratory.

References

[1] Kai Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. Ph.D. Thesis, Yale Univ., TR YALEU-RR-492, Sept. 1986.

[2] P. Keleher, A.L. Cox, S. Dwarkadas, et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proc. of the 1994 Winter USENIX Conference, pages 115-131, Jan. 1994.

[3] John B. Carter. Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency. Ph.D. thesis, Rice Univ., Sept. 1993.

[4] C.K. Shieh, A.C. Lai, J.C. Ueng, et al. Cohesion: An Efficient Distributed Shared Memory System Supporting Multiple Memory Consistency Models. In Proc. of Aizu Int'l Symp. on Parallel Algorithms/Architecture Synthesis, pages 146-152, Feb. 1995.

[5] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Software - Practice and Experience 18(8), Aug. 1988.

[6] Evangelos P. Markatos and Thomas J. Leblanc. Using Processor Affinity in Loop Scheduling on Shared-memory Multiprocessors. in Proc. of Supercomputing '92, pages 104-113, Nov. 1992.

[7] John N. Mellor-Crummey and Michael L. Scott. Synchronization without Contention. In 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, pages 269-278, Apr. 1991.

[8] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, et al. The Amber System: Parallel Programming on a Network of Multiprocessor. In Proc. of the 12th ACM Symp. on Operating System Principles, pages 147-158, Dec. 1989.

[9] iRMK 1.3 Real-Time Kernel Reference Manual. intel Corporation, 1989.

[10] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing. Technical report TR 95-14, The Univ. of Arizona, Dec. 1995.

[11] Dilip Khandekar. Quarks: Portable Distributed Shared Memory on Unix. Technical report in Quarks package, Oct. 1995.

[12] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, et al. Millipede: Easy Parallel Programming in Available Distributed Environments. Technical report LPCR #9506, Technion - Israel Institute of Technology, Nov. 1995.

[13] Tyng-Yeu Liang, Ce-Kuen Shieh, and Weiping Zhu. Task Mapping on Distributed Shared Memory Systems Using Hopfield Neural Network. To appear in Western Multi-Conference '97, Jan. 1997.

[14] An-Chow Lai, Ce-Kuen Shieh, Jyh-Chang Ueng, et al. On the Consideration and Solution for Load Balance in Software Distributed Shared Memory Systems. To appear in the 1997 IEEE Int'l Performance, Computing, and Communications Conf., Feb. 1997.