Atomic Page Update Methods for OpenMP-Aware Software DSM

Yang-Suk Kee
Institute of Computer Technology
Seoul National University
yskee@iris.snu.ac.kr
Jin-Soo Kim
Division of Computer Science
KAIST
jinsoo@cs.kaist.ac.kr
Woo-Chul Jeun, Soonhoi Ha
School of Computer Science and Engineering
Seoul National University
{wcjeun,sha}@iris.snu.ac.kr

Abstract

When software distributed shared memory (SDSM) is extended to utilize threads in support of OpenMP, a challenge is how to preserve memory consistency in a thread-safe way, which is known as "atomic page update problem". In this paper, we show that this problem can be solved by creating two independent access paths to a physical page and by assigning different access permissions to them. Especially, we discuss three new methods using System V shared memory IPC, a new mdup() system call, and a fork() system call as well as a known method using file mapping. The main contribution of this paper is to introduce various solutions to the atomic page update problem and to compare their characteristics extensively. Experiments carried out on a Linux-based cluster of SMPs and an IBM SP Nighthawk system show that the proposed methods achieve better performance than the file mapping method and the method using the process creation mechanism is the best candidate for the IBM SP system.

1. Introduction

OpenMP [1] is becoming the de facto standard for shared-address-space programming model. In addition to programming easiness inherent in shared-address-space model, OpenMP anticipates high performance in scientific applications. Even though the general target architecture of OpenMP is a single multiprocessor node, this model can be applicable to a cluster of multiprocessors. An intuitive way to extend OpenMP to cluster of multiprocessors is to use software distributed shared memory (SDSM), which emulates a shared address space over distributed memories.

Many SDSM systems are implemented at user-level by using the page fault handling mechanisms, assuming uniprocessor nodes. This kind of SDSM system detects an unprivileged access to a shared page by catching a SIGSEGV signal and a user-defined signal handler updates the invalid page with a valid one. From the application point of view, this page-update is atomic since program control is returned to the application only after the signal handler completes the service on the fault.

However, these single-threaded systems are inadequate to the thread-based parallelism of OpenMP. The conventional fault-handling process will fail in multithreaded environments because other threads may try to access the same page during the update period. The SDSM system faces a dilemma when multiple threads compete to access an invalid page within a short interval. On the first access to an invalid page, the system should set the page writable to replace with a valid one. Unfortunately, this change also allows other application threads to access the same page freely. This phenomenon is known as atomic page update and change right problem [2] or mmap() race condition [3]. For short, we call this *the atomic page update problem*.

A known solution to this problem adopted by major multithreaded SDSM systems like TreadMarks [4], Brazos [5], and Strings [6] is to map a file to two different virtual addresses. Even though the systems using file mapping achieve fair good performance on dedicated systems, file mapping is not always the best solution. Operating system and working environment severely affect the performance of these systems. Moreover, file mapping has high initialization cost, experiences buffer

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

caches flushing overhead, and reduces the available address space because SDSM and application partition the address space.

We note the cause of this problem is that SDSM and application share the same address space. When SDSM changes a page writable, the page is also accessible to the application. A general solution to this problem is to separate the application address space from the system address space for the same physical memory, and to assign different access permission to each address space. Then, the system can guarantee the atomic page update by changing the access permission of a virtual page in the application address space only after it completes the page update through the system address space.

In this paper, we present three new solutions using System V shared memory IPC, a new mdup() system call, and a fork() system call as well as a known solution using file mapping. The main contribution of this paper is to present various solutions to the atomic page update problem and to compare their characteristics extensively. However, it is observed that it is not always possible to implement all of them in a given SMP cluster system due to the various limitations of a given operating system. Experiments on a Linux-based cluster and on an IBM SP2 machine show that the proposed methods overcome the drawbacks of the file mapping method such as high initialization cost and buffer cache flushing overhead. Moreover, the method using a fork() system call exploits whole the address space and is a robust method for dynamic environments.

This paper is organized as follows. In section 2, we discuss the atomic page update problem in detail. We briefly introduce our OpenMP-aware SDSM system in section 3 and present four methods to solve the problem in section 4. We investigate four methods by using micro-benchmarks and give experimental results with several applications in section 5. Section 6 concludes the paper.

2. The Atomic Page Update Problem

A typical page fault handling process of conventional page-based SDSM is illustrated in Figure 1. In general, this kind of SDSM uses SIGIO and SIGSEGV signals to implement memory consistency protocols. When the application (T2) accesses the invalid page denoted by A, the operating system generates a SIGSEGV signal and hands over program control to SDSM by invoking a user-defined SIGSEGV handler. Inside the handler, the system allocates a writable page by dynamically creating an anonymous page or by retrieving a page from the shared memory pool prepared in the initialization step. Then, the system requests the most up-to-date page from a remote node and waits for the page. When the page request arrives at the remote node, the remote operating system generates a SIGIO signal and a user-defined SIGIO handler serves the request. After that, the local SDSM replaces the invalid page with the new one and sets the page readable by using an mprotect() system call.

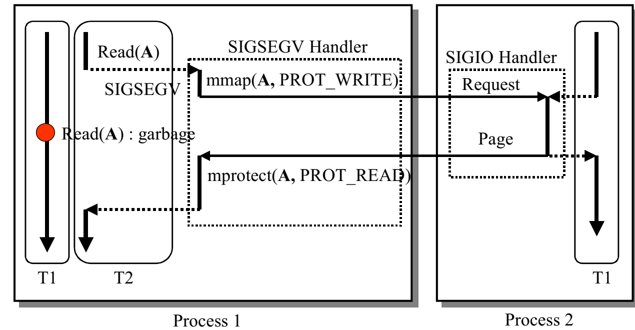

Figure 1. A typical procedure of page fault handling in a conventional page-based SDSM system

In a single-threaded system, this page update is atomic with respect to the application since the program control is returned to the application only after the system completes in replacing the invalid page with a valid one. Atomicity, however, is not guaranteed when multiple threads compete to access a page. Figure 1 illustrates the situation where T1 accesses the same page while T2 is waiting for the up-to-date page after it has set the page writable. T1 continues its computation with garbage data without raising any protection fault. This depicts the atomic page update problem.

A known solution to this problem is to map a file to two virtual addresses and to create two independent access paths to the file: one for application and the other for SDSM. The system can update the file through the virtual address mapped to it while the access from an application thread is controlled by a memory consistency protocol. From the viewpoint of operating system, file mapping is to attach physical pages, used as cache for a file, to the process's virtual address space. When a file is mapped to two virtual addresses, each physical page is pointed by two page table entries and different access permission can be assigned to different virtual addresses. In consequence, the SDSM system guarantees the atomic page update with respect to all application threads by changing the access permission of the virtual pages mapped for application only after it updates the physical pages through the virtual address mapped for system.

A scenario of thread-safe page update in data race by separating the access paths is illustrated in Figure 2. When an application thread tries to access the invalid page denoted by A, SDSM updates the invalid page with the up-to-date page through the system address denoted by S. After the page update is completed, the system changes the page A in the application address space

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

readable and hands over program control to the application thread again. If another threads attempt to access the same page during the update period, they see the page is still invalid and are blocked inside the SIGSEGV handler. When the page update is completed, the signal handler wakes up all the threads waiting for the page.

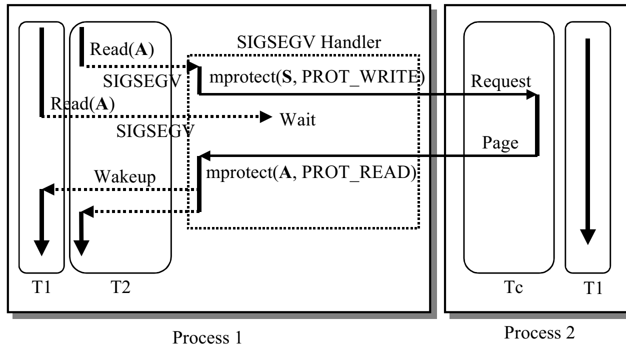

Figure 2. A scenario of the thread-safe page update in data race

File mapping, however, is not the only way to create multiple access paths to a physical page. We seek for other methods to achieve the same goal without performance degradation. In this paper, we propose three more methods and study their characteristics.

3. The ParADE System

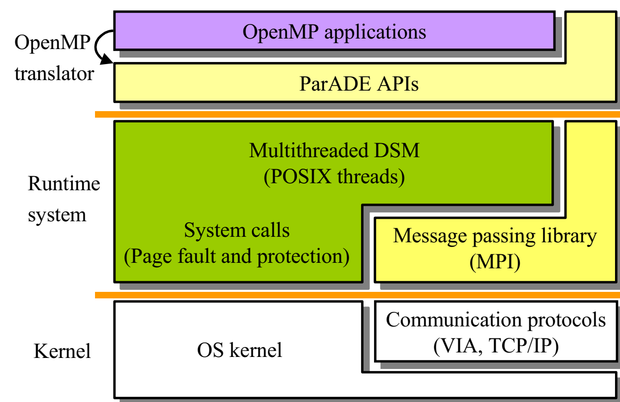

Figure 3. Architecture of the ParADE system

Our SDSM is a component of an OpenMP-based parallel programming environment for SMP clusters called ParADE [7]. Figure 3 depicts the architecture of the ParADE system. Two key components of ParADE are the ParADE runtime system and the OpenMP translator. A multi-threaded SDSM and a message-passing library compose the runtime system. To provide thread-safe communication, we implemented a subset of MPI [8] library for Virtual Interface Architecture (VIA) [9]. We also developed our own SDSM system, which provides a home-based lazy release consistency (HLRC) [10] with migratory home to exploit data locality. Meanwhile, the OpenMP translator converts an OpenMP program to a multi-threaded program with hybrid communication interfaces by using the ParADE runtime library, and enables the program to be executable on the SMP cluster. For more information about ParADE, refer to [7].

4. Four Atomic Page Update Methods

In this section, we present four methods to provide multiple access paths to a physical page: file mapping, System V shared memory IPC, a new mdup() system call, and a fork() system call. All the methods except the mdup() method are implemented at user-level.

4.1. File mapping

An mmap() system call enables a process to access a file through memory operations by mapping the file to the process address space. Moreover, the system call with the MAP_SHARED flag enables a file to be mapped to a process multiple times. Figure 4 illustrates how to make two virtual addresses refer to the same file by mapping a file multiple times.

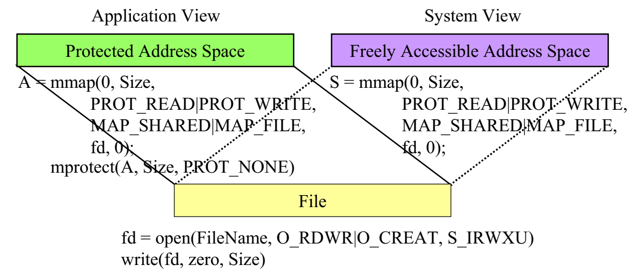

Figure 4. Mapping a file to two virtual addresses

File mapping is very portable and the performance of SDSM with this method is fairly good. Nevertheless, this method has several drawbacks. First, the size of the shared address space should be smaller than the size of the file. When the area beyond the file size is accessed, the operating system signals an error. To avoid this unexpected error, the SDSM system should create a large regular file enough to contain the shared pages or it should dynamically enlarge the file size by explicitly using the write() or ftruncate() operations. However, this initialization cost is not negligible.

1066-6192/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

Another drawback is unnecessary disk writes at runtime. Although FreeBSD supports the MAP_NOSYNC flag to avoid dirty pages to be flushed to disk at runtime, many operating systems flush buffer caches to disk regularly, or explicitly when the munmap() system call is invoked to eliminate the mapping. Disk write is a costly operation so that it may damage performance significantly. In consequence, the performance of a system based on the file mapping method depends on the system buffer cache (page cache) size and the buffer cache management scheme. Experiments on IBM SP Night Hawk system with an AIX 4.3.3 PSSP 3.2 version revealed significant performance degradation when the machine is not wholly dedicated to SDSM.

4.2. System V shared memory IPC

Another method to map a physical page to different virtual addresses is to use System V shared memory IPC. An shmget() system call enables a process to create a shared memory object in the kernel and the shmat() system call enables the process to attach the object to its address space. In addition, shown in Figure 5, a process can attach the shared memory object to its address space more than once and a different virtual address is assigned to each attachment.

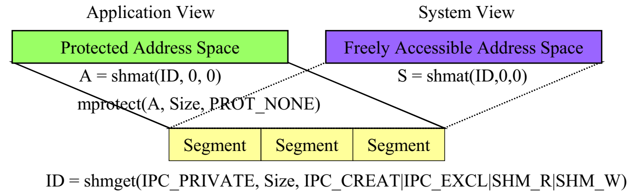

Figure 5. Attaching shared memory segments to two virtual addresses

Compared to file mapping, creating shared memory segments is very cheap. Nevertheless, this mechanism has several restrictions. In some operating systems, the size and the number of shared memory segments are limited. Solaris systems determine the size and the number of segments at boot time by checking the *shmsys* field of the */etc/system* file. In the case of Linux systems, the maximum size of a segment is 32 megabytes and the system-wide maximum number of segments is limited to 128. Some operating systems just allow less than 10 segments whose size should be smaller than tens of kilobytes. As a result, they fail to allocate large shared memory with this method. Moreover, observed in the IBM SP Night Hawk system, the mprotect() system may not be used to change the access permission of shared memory segments allocated by System V shared memory IPC.

Another problem is that a group of segments should be mapped to a continuous address space. When one forces to attach a shared memory segment to a user-assigned address, the attachment will fail if the address is not a predefined address for segment low boundaries. Therefore, we should allocate a segment according to the low boundary address and attach it to a continuous address space. The last consideration is memory leak. Shared memory segments are not released automatically when a program terminates. SDSM should make sure that shared memory segments are released at termination, even at abnormal termination.

4.3. mdup() system call

We implement a new system call, mdup(), to easily duplicate the per-process page table. The prototype of mdup() is as follows.

void mdup(void* addr, int size)*,

where *addr* is the virtual address of the anonymous memory region created by the mmap() system call with the MAP_ANONYMOUS and MAP_SHARED flags and *size* is the size of the region.

The basic mechanism of mdup() is to allocate new page table entries for the detour and to copy the page table entries of the anonymous memory to new ones. The reasons why we use anonymous memory are following: (1) no initialization step is required and (2) there is no size limit. Even though kernel modification damages portability of SDSM, the mdup() system call is easy to use and overcomes many drawbacks of the previous methods.

4.4. fork() system call

The total amount of physical memory in a cluster system increases with the size of cluster. Nevertheless, the size of the virtual address space is fixed and puts restriction on the problem size of applications. The previous methods reduce the virtual address space available for applications because the application and the system partition the address space. Therefore, we propose another method to support thread-safe memory management without sacrificing the address space.

When a process forks a child process, the child process inherits the execution image of the parent process. The parent process creates shared memory regions and forks a child process. Then, they have independent access paths even though they use the same virtual address to access the same physical page. We let the parent process execute applications and the child process perform memory consistency mechanisms. Hence, the SDSM system can successfully update the shared memory region in a thread-safe way through the child process's address space.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

Operations	Linux Pentium III 600Mhz		IBM SP POWER3 375Mhz	
	4 KB	64 MB	4 KB	64 MB
mmap()-file mapping	5.0	35.9	21.2	88.9
mmap()-anonymous memory	6.5	43.9	19.4	82.7
shmget()	7.2	54.7	10.4	57.0
shmat()	4.9	31.4	6.7	25.4
mdup()	316.6	1720.7	N/A	N/A
fork()	94.0	17348.8	2998.7	5777.1
write()	43.6	849617.2	47.6	865767.7
mprotect()-file mapping	3.4	37.5	10.9	40074.4
mprotect()-anonymous memory	2.9	33.7	9.4	20.7
mprotect()-System V shared memory	4.4	34.1	N/A	N/A
memcpy()-file mapping	5.0	472543.6	16.2	1371498.1
memcpy()-anonymous memory	7.8	492053.5	32.6	659901.3
memcpy()-System V shared memory	7.9	530368.3	27.1	499294.0
SIGSEGV handler	9.8		10.2	
munmap()-file mapping	5.7	17117.9	19.1	108993.7
munmap()-anonymous memory	10.2	46934.5	27.1	174688.1
shmdt()	28.0	14528.1	6.0	30.2
shmctrl()	8.6	30821.6	16.8	110888.6

Table 1. Costs of basic operations (us)

However, this method experiences additional latency due to communication and synchronization overheads between the parent and the child processes. Nonetheless, this method is very portable and it survives even under a harsh working environment like IBM SP Night Hawk.

5. Experiments

We have implemented four methods in the ParADE runtime system. We first measured the costs of basic operations and compared the performance of the methods with several applications. Our experiments were performed on an IBM SP Night Hawk system and a Linux cluster. The IBM SP system consists of nine 375Mhz POWER3 SMP nodes with sixteen processors and 16GB main memory per node. The Linux cluster consists of four dual-Pentium III 550Mhz SMP nodes and four dual-Pentium III 600Mhz SMP nodes. Each node has 512 MB main memory and it is connected to a Giganet's cLAN VIA switch. Redhat 8.0 with a kernel of 2.4.18-14 SMP version runs on each node. We used a GNU gcc compiler with the -O2 option for Linux cluster and an xlc complier with the -O2 -qarch=pwr3 -qtune=pwr3 -qmaxmem=-1 -qstrict options for the IBM SP system.

5.1. Costs of basic operations

Table 1 shows the costs of the basic operations used by four methods. The operations in the top group are used in the initialization step, those in the middle are used at runtime, and those in the bottom are used at finalization. We take the average execution time after 100 executions of micro-benchmark programs.

Since the top operations are used to create a shared memory pool, the execution time for handling large memory is important. Note that creating a 64-megabyte file is very expensive compared to System V shared memory and anonymous memory. The main difference between file mapping and the others is the time of actual memory allocation. In the case of file mapping, physical pages are allocated at the initialization step in the form of buffer cache or page cache. However, the other methods delay the page allocation until a page is actually referenced at runtime.

Since the page size of both operating systems is 4 kilobytes, the costs for the operations handling 4 kilobytes memory are important at runtime. The cost of memcpy() operation for the mapped file is lower than that for the other methods. The shorter elapsed time mainly stems from the fact that the other methods experience additional memory allocation overhead. However, the results with 64-megabyte memory are different. For file mapping on the IBM SP machine, the copy operation suffers from long latency because of buffer cache flushing overhead.

To understand how these basic operations affect the system actually at runtime, we analyze the page fetch latency. Figure 6 shows the factors in fetching a page from a remote node on a read fault on two dual-Pentium III 600 MHz nodes. To avoid caching effect, we allocate

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

Application	Input size	Declared shared memory (MB)
CG	A-class	64
Helmholtz	1000 x 1000 matrix	32
MD	1000 iterations	1

Table 2. Application characteristics

Application	File mapping	System V shared memory	mdup()	fork()
CG	0.891	0.002	0.002	0.001
Helmholtz	0.446	0.001	0.002	0.002
MD	0.015	0.001	0.002	0.001

Table 3. Initialization costs on a dual-Pentium III 600 MHz node (s)

large shared memory and measure the page fetch latency changing the accessing points in the shared memory area. Executing the SIGSEGV handler (Fault handler call) and sending a page request to the home node (Request transfer) are independent of the methods. In the case of protocol overhead, the fork() method experiences about twice longer latency than the others due to inter-process communication overhead between the parent and the child processes. The page preparation and page transfer factors are dependent on the methods. However, as shown in Table 1, all the methods have comparable performance of the mprotect() and the memcpy() system calls with 4 kilobyte page. Figure 6 shows the similar result that these two factors little influence on the total page fetch latency regardless of the methods.

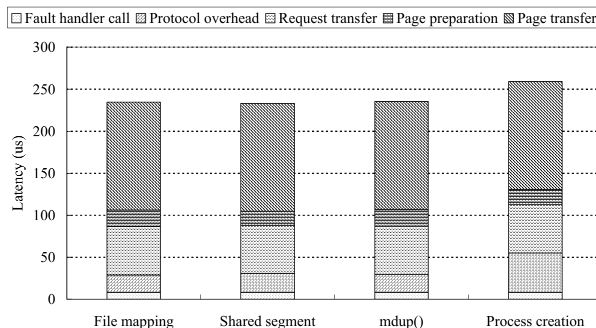

Figure 6. Page fetch latency on two Pentium III 600Mhz nodes (us)

5.2. Application performance

We compare the performance of four methods by measuring the execution time of the NPB CG kernel [11] and two real applications [12,13]. We ported the Fortran programs to the C versions. We take the average execution time after 10 executions of the programs. Only the results on the Linux cluster are presented because the System V shared memory method and the mdup() system call cannot be implemented in the SP system and the file mapping method reveals extremely long execution time due to the high memory copy overhead in a shared working environment. We use the *vmstat* command to monitor the system dynamics.

The characteristics of the programs and the initialization costs of the methods are shown in Table 2 and Table 3, respectively. The CG and the Helmholtz programs have large shared memory while the MD program has small one. As shown in Table 1, the initialization cost of the file mapping method with large shared memory is very expensive. In case an application has relatively short execution time, this high initialization cost can be critical to overall performance. However, applications with small shared memory are little influenced by the initialization cost regardless of methods.

One fundamental question about the atomic page update problem is whether it is serious in real applications. Figure 7 shows the ratio of the number of faults in the racing condition to the number of total read faults. It demonstrates that the atomic page problem is common and it is dependent on the computing pattern, not on the amount of shared memory.

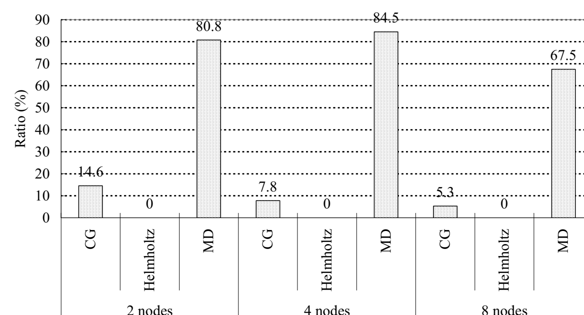

Figure 7. Ratio of the number of racing faults to the number of read faults

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

Figure 8 shows the execution time of the CG kernel of A class varying the number of nodes. With respect to the overall execution time, file mapping shows the worst performance though the performance difference is not huge. To understand the performance of file mapping, we monitor the number of block transfers. At the initialization step, over fifteen thousand blocks are read from disk and over one hundred thousand blocks are written to disk. However, only about one hundred blocks are written to disk at runtime. Therefore, the disk-write penalty affects the system severely at the initialization step but little at runtime. This phenomenon occurs consistently regardless of the number of nodes. As the portion of CPU resource assigned to communication increases with the number of nodes, the performance with 8 nodes becomes worse than with 4 nodes.

In the case of MD, the size of shared memory is only about 1 megabyte and the initialization cost does not affect the overall performance severely. The net execution time of file mapping is a few seconds longer than the others but it is hardly noticeable in Figure 9. Meanwhile, Helmholtz requires 32 megabytes shared memory but the initialization cost is amortized over the computation. One interesting result in Figure 10 is that the process creation method achieves the best performance. In fact, the number of context switchings of the process creation method is twice of the number of the others. However, other system dynamics overwhelm the context-switching overhead.

6. Conclusions

In this paper, we presented four methods to solve the atomic page update problem and studied their characteristics extensively. Experiments on a Linux based cluster and on an IBM SP2 machine showed that the three proposed methods overcome the drawbacks of the file mapping method such as high initialization cost and buffer cache flushing overhead. In particular, the method using a fork() system call is portable and preserves the whole address space to the application even though the others can use only the half of the virtual address space. The System V shared memory method shows low initialization cost and runtime overhead, and the new mdup() system call method has the least coding overhead in the application code. Not all the methods can be implemented on a given SMP cluster system due to the limitations of the operating system as observed in the IBM SP System. The methods proposed for thread-safe memory management will allow us to port the ParADE environment to various systems.

7. Acknowledgements

This work was supported by National Research Laboratory Program (No. M1-0104-00-0015) and Brain Korea 21 Project. The ICT at Seoul National University provides research facilities for this study.

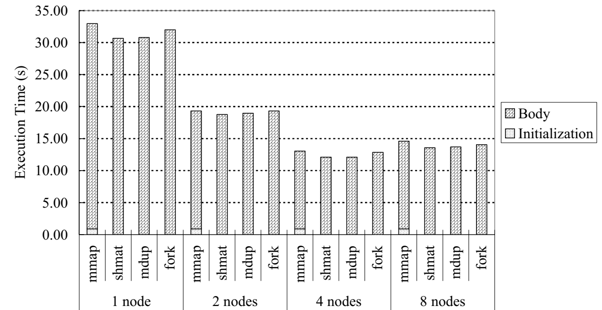

Figure 8. Execution time of CG of A class using two processors

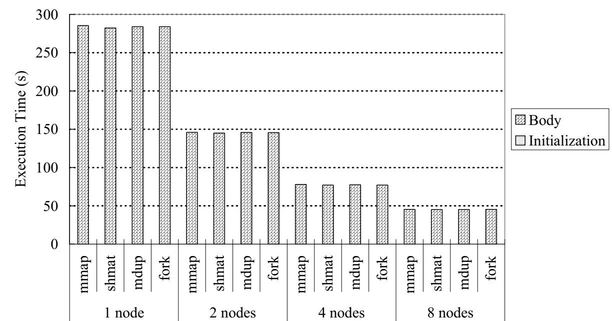

Figure 9. Execution time of MD using two processors

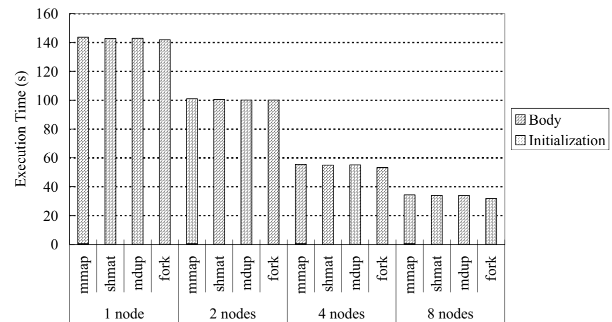

Figure 10. Execution time of Helmholtz using two processors

8. References

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.

[1] OpenMP C and C++ Application Programming Interface, Version 1.0, http://www.openmp.org, Oct. 1998.
[2] F. Mueller. ”Distributed Shared-Memory Threads: DSM-Threads”. Workshop on RunTime systems for Parallel Programming, pp. 31–40, Apr. 1997.
[3] M. Pizka and C. Rehn, ”Murks-A POSIX Threads Based DSM System”, In the proceedings of the international conference on Parallel and Distributed Computing Systems, 2001.
[4] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel, ”OpenMP for Networks of SMPs,” Journal of Parallel and Distributed Computing, vol. 60, no.12, Dec. 2000, pp. 1512-1530.
[5] Evan Speight and John K. Bennett, ”Brazos: A Third Generation DSM System”, USENIXWindows NT Workshop, Aug. 1997, pp. 95-106.
[6] Sumit Roy and Vipin Chaudhary, ”Strings: A High-Performance Distributed Shared Memory for Symmetric Multiprocessor Clusters”, International Symposium on High Performance Distributed Computing, July 1998, pp. 90-97.
[7] Yang-Suk Kee, Jin-Soo Kim, Soonhoi Ha, “ParADE: An OpenMP Programming Environment for SMP Cluster Systems”, Proceedings of ACM/IEEE Supercomputing (SC'03), Nov. 2003
[8] Message-passing Interface Forum, "MPI: A message-passing interface standard," International Journal of Supercomputer Applications and High Performance Computing, 8(3/4), pp. 159-416, 1994.
[9] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, Chris Dodd, ”The Virtual Interface Architecture,” IEEE Micro, vol. 18, no. 2, Mar./Apr. 1998, pp. 66-76.
[10] L. Iftode. ”Home-based Shared Virtual Memory”. (PhD thesis), 1998.
[11] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow, ”The NAS Parallel Benchmarks”, Report NAS-95-020, 1995, http://www.nas.nasa.gov/Software/NPB.
[12] Joseph Robicheaux, http://www.openmp.org/samples/jacobi.f, 1998.
[13] Bill Magro, Kuck, and Associates, http://www.openmp.org/samples/md.f, 1998.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:55 from IEEE Xplore. Restrictions apply.