ELSEVIER

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Parallel Computing 30 (2004) 121–138

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Memory management for multi-threaded software DSM systems

Yang-Suk Kee [a,*], Jin-Soo Kim [b], Soonhoi Ha [a]

[a] *School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, South Korea*

[b] *Division of Computer Science, KAIST, Daejeon 305-701, South Korea*

Received 22 January 2003; received in revised form 25 August 2003; accepted 2 September 2003

Abstract

When software distributed shared memory (SDSM) systems provide multithreading to exploit cluster of symmetric multiprocessors (SMPs), 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 propose three new methods using System V shared memory inter-process communication (IPC), a new mdup() system call, and a fork() system call in addition to 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 Night Hawk system show that the 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 preserves the whole address space to the application.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Distributed shared memory; Atomic page update; Memory consistency; Cluster of symmetric multiprocessors

* Corresponding author. Fax: +82-2-879-1532.
E-mail addresses: yskee@iris.snu.ac.kr (Y.-S. Kee), jinsoo@cs.kaist.ac.kr (J.-S. Kim), sha@iris.snu.ac.kr (S. Ha).

0167-8191/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2003.09.001

1. Introduction

Software distributed shared memory (SDSM) systems have been powerful platforms to provide shared address space on distributed memory architectures. The early SDSM systems like IVY [1], Midway [2], Munin [3], and TreadMarks [4] assume uniprocessor nodes, thus allow only one thread per process on a node. Currently, commodity off-the-shelf microprocessors and network components are widely used as building blocks for parallel computers. This trend has made cluster systems consisting of symmetric multiprocessors (SMPs) attractive platforms for high performance computing. However, the early single-threaded SDSM systems are too restricted to exploit multiprocessors in SMP clusters. The next generation SDSM systems like Quarks [5], Brazos [6], DSM-Threads [7], and Murks [8] are aware of multiprocessors and exploit them by means of multiprocesses or multithreads. In general, naive process-based systems experience high context switching overhead and additional inter-process communication delay within a node, so our focus in this paper is on multi-threaded SDSM systems.

Many single-threaded SDSM systems are implemented at user-level by using the page fault handling mechanisms. This kind of SDSM detects an unprivileged access of application 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, 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 [7] or mmap() race condition [8]. For short, we call this *the atomic page update problem*.

A known solution to this problem adopted by major multithreaded SDSM systems like TreadMarks [9], Brazos [6], and Strings [10] is to map a file to two different virtual addresses. Even though the file mapping method achieves good performance on some systems, file mapping is not always the best solution. Operating system and working environment may severely affect the performance of these systems. The file mapping method performs poorly in some cases; for example, an IBM SP Night Hawk system with AIX 4.3.3 PSSP 3.2 version. This observation motivates the research of other solutions to the atomic page update problem. Moreover, file mapping has high initialization cost and reduces the available address space because SDSM and application partition the address space.

We note the cause of the atomic page update 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. Since the virtual memory protection mechanism is implemented in the per-process page table, different virtual addresses (pages) can have different access permission even though they refer to the same physical page. 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 propose three new solutions using System V shared memory inter-process communication (IPC), a new mdup() system call for page table duplication, and a fork() system call in addition to 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 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 overhead and buffer cache flushing overhead. In particular, the method using a fork() system call preserves the whole address space to the application.

This paper is organized as follows. In Section 2, we discuss the atomic page update problem in detail. We briefly introduce our 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 Fig. 1. In general, this kind of SDSM uses SIGIO and SIGSEGV signals to implement memory consistency protocols. When the application accesses the invalid

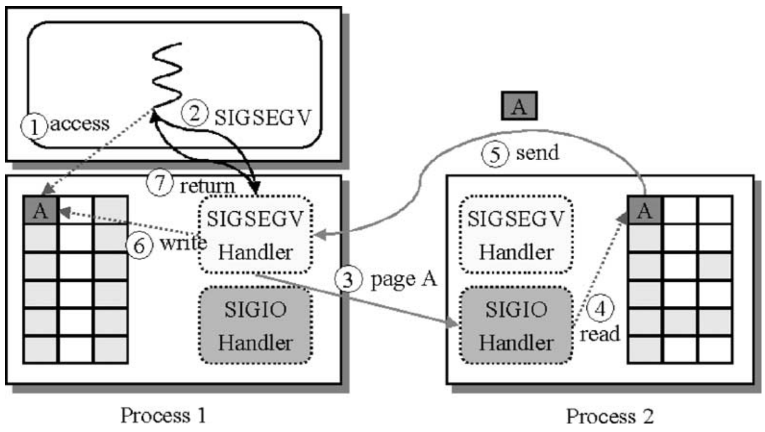


Fig. 1. A typical procedure of page fault handling in a conventional page-based SDSM system.

page denoted by A, the operating system generates a SIGSEGV signal and hands over the 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 using an mprotect() system call.

In a single-threaded system, this page update is atomic with respect to the application since the system performs the series of operations sequentially. Atomicity, however, is not guaranteed when multiple threads compete to access a page. Fig. 2 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.

In order to guarantee the atomic page update, other threads should be prevented from accessing the page while a thread is waiting for a valid page. Several possible solutions can be categorized as follows:

- Suspend all the application threads until the system finishes updating the invalid page.
- Modify the OS scheduler not to schedule the threads that may access the invalid page [8].
- Implement a new thread package [5,7].
- Map a file to two virtual addresses and assign different access permission to them [6,9,10].

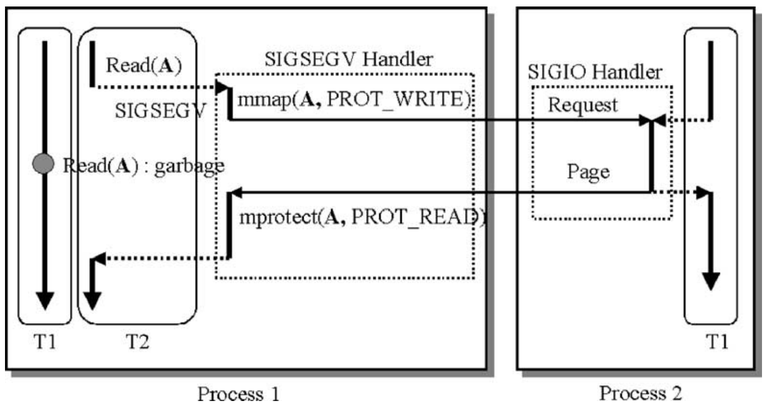


Fig. 2. The atomic page update problem in a conventional page-based SDSM system.

The first brute-force method suspends all application threads temporarily by broadcasting a SIGSTOP signal during the page update period and awakes the threads again after the page update is completed. This method is very simple but it obstructs even the execution of the threads unrelated to the page. Consequently, this method degrades the CPU utilization prohibitively and the expected performance is poor. The second approach is to modify the OS kernel to schedule only the threads that do not access the same page. Murks [8] is this type of system. Even though this approach is efficient, it damages portability of the system. The third one is to implement a new thread package to control thread scheduling at the user-level. DSM-Threads [7] and Quarks [5] are two well-known systems but they lack portability.

The last method 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.

Specifically, we notice the last method. The key point of file mapping is to create two independent access paths to a physical page. A scenario of thread-safe page update in data race by separating the access paths is illustrated in Fig. 3. 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.

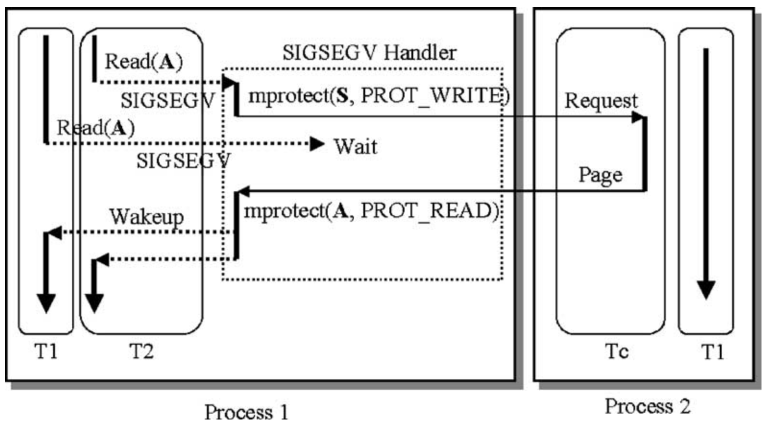


Fig. 3. A scenario of the thread-safe page update.

After the page update is completed, the system changes the page A in the application address space readable and hands over the program control to the application thread again. If another thread attempts to access the same page during the update period, it sees the page is still invalid and is blocked inside the SIGSEGV handler. When the page update is completed, the signal handler wakes up all the threads waiting for the page.

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 compare their characteristics.

3. The ParADE system

Our SDSM is a component of an OpenMP-based parallel programming environment for SMP clusters called ParADE [11]. OpenMP [12] 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 is applicable to a cluster of multiprocessors. An intuitive way to extend OpenMP to cluster of multiprocessors is to use a multi-threaded SDSM system.

Fig. 4 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 [13] library for Virtual Interface Architecture (VIA) [14]. Our SDSM system provides a home-based lazy release consistency (HLRC) [15] with migratory home to exploit data locality. Meanwhile, the OpenMP translator converts an OpenMP program to a multi-

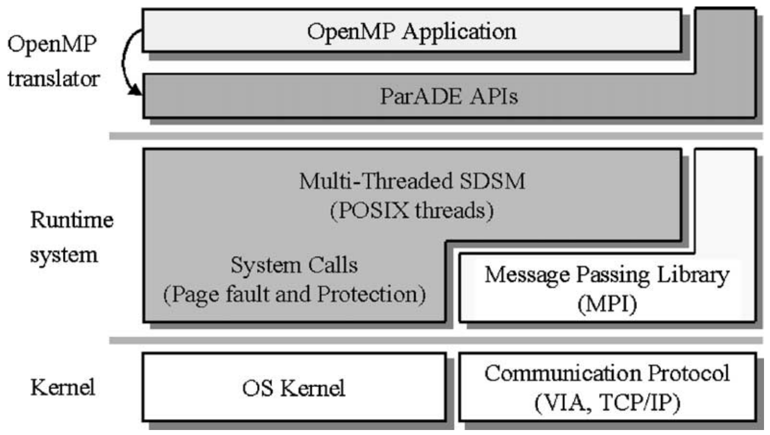

OpenMP translator
OpenMP Application
ParADE APIs
Runtime system
Multi-Threaded SDSM (POSIX threads)
System Calls (Page fault and Protection)
Message Passing Library (MPI)
Kernel
OS Kernel
Communication Protocol (VIA, TCP/IP)

Fig. 4. Architecture of the ParADE system.

threaded program with hybrid communication interfaces using the ParADE runtime library, and enables the program to be executable on the SMP cluster. For more information about ParADE, refer to [11].

4. Four atomic page update methods

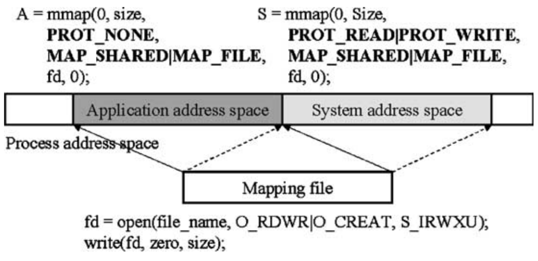
In this section, we present four methods to provide multiple access paths to a physical page: file mapping, System V shared memory IPC, a new mdup() system call, and a fork() system call. All the methods except the mdup() method are implemented at user-level.

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. This mechanism is known as memory-mapped I/O. Such a mapping creates a one-to-one correspondence between data in the file and data in the mapped memory region. Moreover, the system call with the MAP_SHARED flag enables a file to be mapped to a process multiple times. As shown in Fig. 5, multiple virtual addresses refer to the same file when the file is mapped multiple times, creating multiple independent access paths to a same physical page.

File mapping is very portable and the performance of SDSM using this method is 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. Nonetheless, this initialization cost is not negligible.

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


```
A = mmap(0, size,
    PROT_NONE,
    MAP_SHARED|MAP_FILE,
    fd, 0);

S = mmap(0, Size,
    PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_FILE,
    fd, 0);

fd = open(file_name, O_RDWR|O_CREAT, S_IRWXU);
write(fd, zero, size);
```

Fig. 5. Mapping a file to two virtual addresses.

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

Another method to map a physical page to different virtual addresses is to use System V shared memory IPC. The 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. Meanwhile, as shown in Fig. 6, 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.

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 using this method. Moreover, in the IBM SP Night Hawk system, mprotect() may not be used to change the access permission of System V shared memory segments.

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 process page table. The prototype of mdup() is as follows:

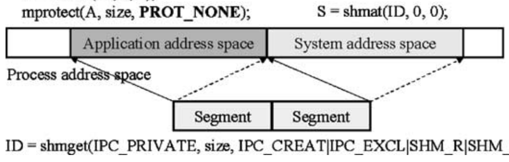

A = shmat(ID, 0, 0);
mprotect(A, size, PROT_NONE);
S = shmat(ID, 0, 0);
Application address space
System address space
Process address space
Segment
Segment
ID = shmget(IPC_PRIVATE, size, IPC_CREAT|IPC_EXCL|SHM_R|SHM_W);

Fig. 6. Attachment of shared memory segments to two virtual addresses.

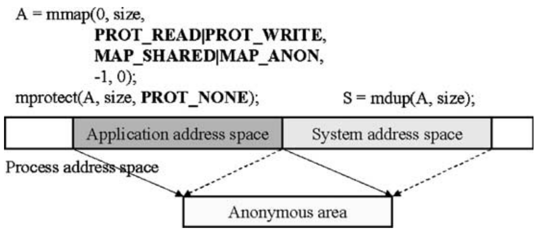


Fig. 7. Duplication of the per-process page table using the mdup() system call.

$$\text{void } *\text{mdup(void } *\text{addr, int size)},$$

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 mdup() method is illustrated in Fig. 7. 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, (2) there is no size limit, and (3) the memory region is released automatically at program termination. 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. 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. Especially, the content of the child process page table is copied from that of the parent process except the data area where a Copy-On-Write policy is applied. However, the Copy-On-Write policy is not applied to shared memory regions. The parent process creates shared memory regions and forks a child process. As shown in Fig. 8, 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.

However, this method experiences additional latency due to communication and synchronization overheads between the parent and the child processes. Nonetheless,

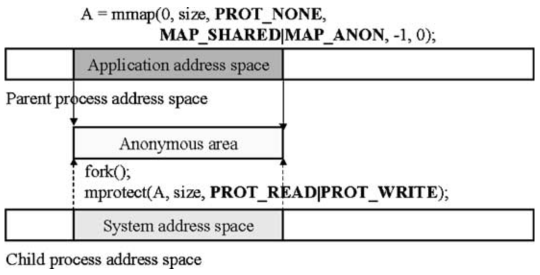

A = mmap(0, size, PROT_NONE, MAP_SHARED|MAP_ANON, -1, 0);
Application address space
Parent process address space
Anonymous area
fork();
mprotect(A, size, PROT_READ|PROT_WRITE);
System address space
Child process address space

Fig. 8. Duplication of the per-process page table using the fork() system call without sacrificing the available address space.

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 375 MHz POWER3 SMP nodes with 16 processors and 16 GB main memory per node. The Linux cluster consists of four dual-Pentium III 550 MHz SMP nodes and four dual-Pentium III 600 MHz 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. Architecture of the ParADE multi-threaded SDSM

A skeleton of the ParADE multi-threaded SDSM is shown in Fig. 9. The left configuration corresponds to the three methods of the file mapping, System V shared memory IPC, and the mdup() system call. In the beginning, the system allocates a virtual shared memory pool and initializes its access permission. The SIGSEGV handler is an entry point to the SDSM system from the application and fetches the up-to-date page from the home node. Asynchronous Message Server thread in the home node serves the page request. The right one is for the fork() method. Unix domain socket is used for control message exchange and synchronization between the parent and the child processes. For communication, the parent creates one socket per application thread and the child process forks a thread called Request Server to handle requests from the parent. An application thread sends a control message to the child process through the socket and waits for a

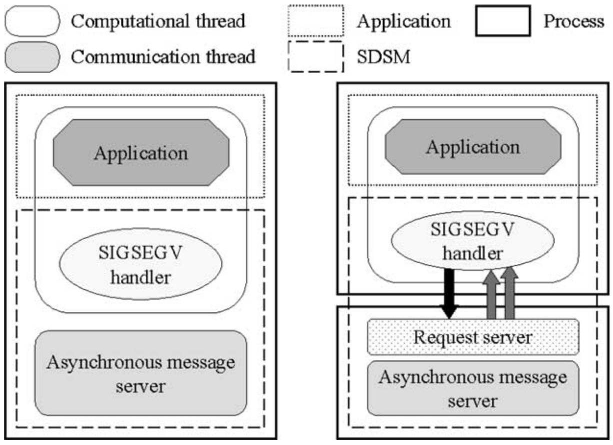


Fig. 9. Skeleton of prototype system: the left configuration is for the file mapping, System V shared memory, and mdup() methods and the right configuration is for the fork() method.

reply from the child process on this socket. When the requested page arrives at the child, Request Server wakes up the application thread.

5.2. 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. In the IBM SP Night Hawk system, the mprotect() system call is not allowed to change the access permission of the memory area allocated by System V shared memory IPC, so the corresponding fields are omitted.

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 megabytes 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 for 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 the memcpy() operation for the mapped file is a little bit lower than that for the other methods. The longer elapsed time mainly stems from the fact that the other methods experience additional memory allocation overhead. However, the results with 64 megabytes memory are different. For file mapping on the IBM SP machine, the copy operation suffers from long latency because of high buffer cache flush overhead.

Table 1
Costs of basic operations (μs)

Operations	Linux Pentium III 600 MHz		IBM SP POWER3 375 MHz	
	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
shmctl()	8.6	30821.6	16.8	110888.6

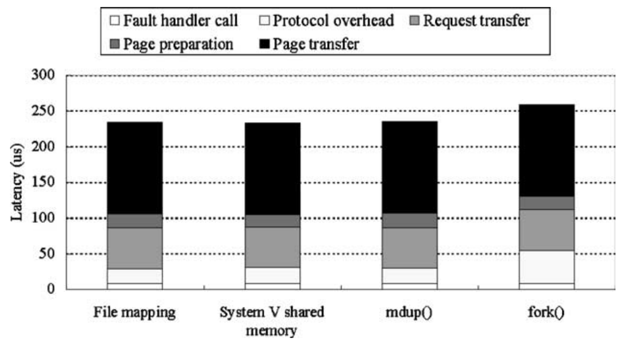


Fig. 10. Analysis of page fetch latency on two dual-Pentium III 600 MHz nodes (μs).

To understand how these basic operations affect the system actually at runtime, we analyze the page fetch latency. Fig. 10 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 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 also dependent on the methods. However, shown in Table 1, all the methods have comparable performance of the mprotect() and the memcpy() system calls with 4 kilobyte page. Fig. 10 shows the similar result that these two factors little influence on the total page fetch latency regardless of the methods.

5.3. Application performance

We compare the performance of four methods by measuring the execution time of several programs. The CG and EP kernels of class-A are adopted from the NAS 2.3 benchmarks [16]. The CG kernel solves an unstructured sparse linear system by the conjugate gradient method and the EP kernel measures the capability of floating-point operations. Meanwhile, two real applications are adopted from the OpenMP sample programs. The Helmholtz program [17] solves a wave equation on a regular mesh using an iterative Jacobi method with over-relaxation and the MD program [18] implements a simple molecular dynamics simulation in continuous real space. 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 high memory copy overhead in a shared working environment. We use the *vmstat* command to monitor the system dynamics roughly.

The characteristics of the programs and the initialization costs of the methods are shown in Tables 2 and 3 respectively. The CG and the Helmholtz programs have large shared memory while the EP and the MD programs have 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.

Table 2
Application characteristics

Application	Input size	Declared shared memory (MB)
CG	A-Class	64
EP	A-Class	0
Helmholtz	1000×1000 matrix	32
MD	1000 iterations	1

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

Application	File mapping	System V shared memory	mdup()	fork()
CG	0.891	0.002	0.002	0.001
EP	0.000	0.000	0.000	0.000
Helmholtz	0.446	0.001	0.002	0.002
MD	0.015	0.001	0.002	0.001

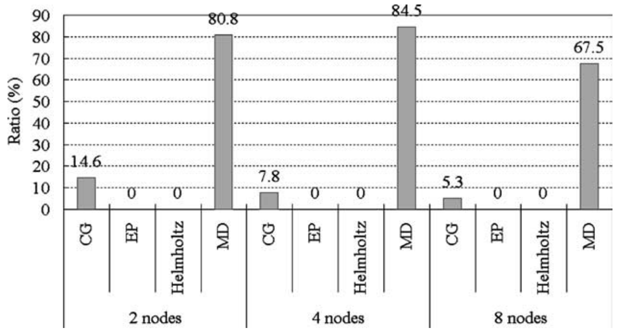


Fig. 11. Ratio of the number of faults in data race to the number of read faults.

One fundamental question about the atomic page update problem is whether it is serious in real applications. Fig. 11 shows the ratio of the number of faults in the racing condition to the number of total read faults. It reveals that the atomic page problem is common and it is dependent on the computing pattern, not on the amount of shared memory.

Fig. 12 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 per-

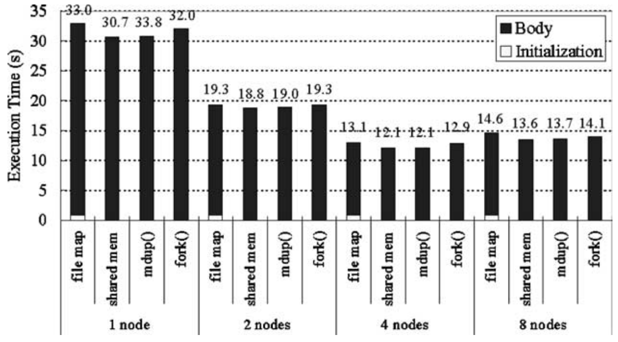


Fig. 12. Execution time of CG of A class using two computational threads on a Linux cluster.

formance of file mapping, we monitor the number of block transfers with the single node configuration. At the initialization step, over 15,000 blocks are read from disk and over 100,000 blocks are written to disk. However, only about 100 blocks are written to disk at runtime. The disk-write penalty affects the system severely at the initialization step but little at runtime. Similar 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 that with 4 nodes. In the case of EP, there is little shared memory, so the file mapping method does not experience initialization overhead. As shown in Fig. 13, all the methods achieve comparable performance.

In the case of MD, the size of shared memory is only about 1 megabyte and the initialization cost rarely affects 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 Fig. 14. Meanwhile, Helmholtz requires 32 megabytes shared memory but the

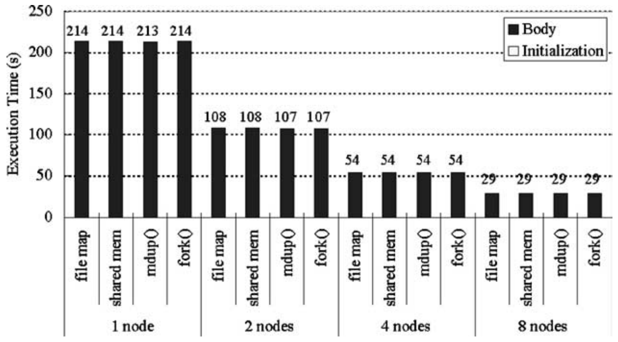


Fig. 13. Execution time of EP of A class using two computational threads on a Linux cluster.

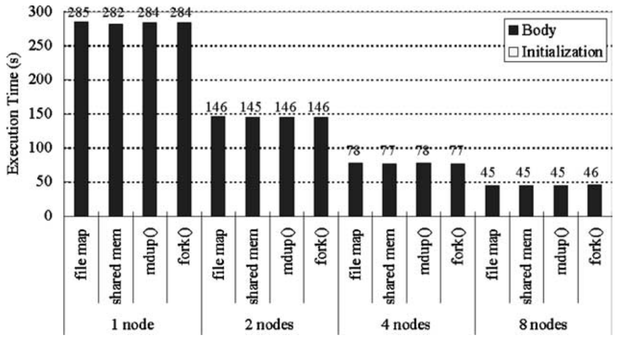


Fig. 14. Execution time of MD using two computational threads on a Linux cluster.

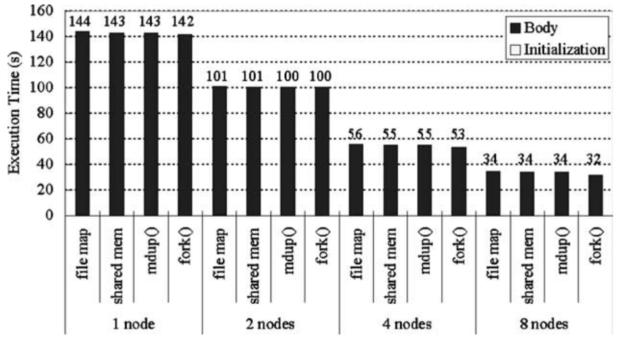


Fig. 15. Execution time of Helmholtz using two computational threads on a Linux cluster.

initialization cost is amortized over the long computation. As shown in Fig. 15, the difference is not significant.

5.4. Implementation cost

We implemented four methods with the following unified interfaces:

createGlobalHeap Creates a regular file or several System V shared memory segments to contain a shared memory pool.
initAppArea Prepares application address space.
initSysArea Prepares system address space.

All require similar amount of codes to implement in spite of the difference in detailed mechanisms. In the createGlobalHeap routine, the file mapping method opens a file and initializes it with zero. The System V shared memory method creates a series of shared memory segments according to the maximum size of a segment. The mmap() and the shmat() system calls are used to prepare the application address space and the mmap(), shmat(), and mdup() system calls are used to make the system address space.

Since all the methods except the fork() method are implemented in a process, the application and the SDSM system share the same data structures. In the case of the fork() method, however, all the shared data structures between the parent and the child should be identified and they should be located in a system-defined shared area. In addition, when MPI library is used for the inter-node communication, communication must be isolated to the child process because most MPI libraries are based on version 1 that does not support dynamic process creation.

6. Conclusions

In this paper, we present four methods to solve the atomic page update problem and compare their characteristics extensively. Table 4 summarizes their characteris-

Table 4
Summary of four atomic page update methods

		File mapping	System V shared memory	mdup()	fork()
System calls	Application path	open()+write() mmap()	shmget() shmgat()	mmap()	mmap()
	System path	mmap()	shmat()	mdup()	fork()
Initialization cost		Very expensive	Very cheap	Cheap	Cheap
Portability		Excellent	Limits in size and number of segments	Poor	Excellent
Address space		Partially available			Fully available
Miscellaneous		Disk write penalty	Segment clean-up cost mprotect() constraint	–	IPC delay between parent and child
Performance		Comparable			
Environment		Dedicated	Shared		

tics with respect to performance, portability, and properties. Experiments on a Linux-based cluster and on an IBM SP2 machine show 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 limitation 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.

Acknowledgements

This work was supported by National Research Laboratory Program (No. M1-104-00-0015) and Brain Korea 21 Project. The ICT at Seoul National University provides research facilities for this study.

References

[1] L. Kai, IVY: a shared virtual memory system for parallel computing, International Conference on Parallel Processing (1988) 94–101.

[2] B.N. Bershad, M.J. Zekauskas, W.A. Sawdon, The midway distributed shared memory system, IEEE International Computer Conference, February 1993, pp. 528–537.
[3] J.K. Bennett, J.B. Carter, W. Zwaenepoel, Munin: distributed shared memory based on type-specific memory coherence, Principles and Practice of Parallel Programming (1990) 168–176.
[4] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: shared memory computing on networks of workstations, IEEE Computer 29 (2) (1996) 18–28.
[5] D.R. Khandekar, Quarks: distributed shared memory as a basic building block for complex parallel and distributed systems, Master's Thesis, University of Utah, 1996.
[6] E. Speight, J.K. Bennett, Brazos: a third generation DSM system, USENIX Windows NT Workshop, August 1997, pp. 95–106.
[7] F. Mueller, Distributed shared-memory threads: DSM-Threads, Workshop on RunTime systems for Parallel Programming, April 1997, pp. 31–40.
[8] M. Pizka, C. Rehn, Murks—A POSIX threads based DSM system, in: Proceedings of the International Conference on Parallel and Distributed Computing Systems, 2001.
[9] Y.C. Hu, H. Lu, A.L. Cox, W. Zwaenepoel, OpensMP for networks of SMPs, Journal of Parallel and Distributed Computing 60 (12) (2000) 1512–1530.
[10] S. Roy, V. Chaudhary, Strings: a high-performance distributed shared memory for symmetric multiprocessor clusters, in: International Symposium on High Performance Distributed Computing, July 1998, pp. 90–97.
[11] Y.-S. Kee, J.-S. Kim, S. Ha, ParADE: an OpenMP programming environment for SMP cluster systems, in: Proceedings of ACM/IEEE Supercomputing, November 2003.
[12] OpenMP C and C++ Application Programming Interface, Version 1.0, October 1998. Available from <http://www.openmp.org>.
[13] Message-passing Interface Forum, MPI: a message-passing interface standard, International Journal of Supercomputer Applications and High Performance Computing, 8 (3–4) (1994) 159–416.
[14] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Marie Merritt, Ed. Gronke, C. Dodd, The virtual interface architecture, IEEE Micro 18 (2) (1998) 66–76.
[15] L. Iftode, Home-based shared virtual memory, Ph.D. Thesis, Princeton University, 1998.
[16] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, The NAS Parallel Benchmarks, Report NAS-95-020, 1995. Available from <http://www.nas.nasa.gov/Software/NPB>.
[17] J. Robicheaux, 1998. Available from <http://www.openmp.org/samples/jacobi.f>.
[18] B. Magro et al., 1998. Available from <http://www.openmp.org/samples/md.f>.