OpenMP on the FDSM software distributed shared memory

Hiroya Matsuba Yutaka Ishikawa

Graduate School of Information Science and Technology
The University of Tokyo

Abstract

This paper describes an OpenMP ready distributed shared memory system called FDSM. FDSM analyzes the access pattern to the shared memory at the first iteration of a loop and obtain the communication set. By using this information, it reduces the overhead of the coherence maintenance. OpenMP on FDSM is evaluated by using the benchmark application CG in the NAS parallel benchmark and compared with another DSM system. The benchmark result shows FDSM runs 40% to 80% faster than another DSM system and 70 % to 90 % slower than the same benchmark written in MPI.

1 Introduction

There are some OpenMP systems running on a distributed memory parallel computer[3, 6]. in which a software distributed shared memory system is used.

The performance of DSM is limited due to the high overhead of the coherency maintenance. FDSM, proposed in this paper, is a new distributed shared memory system, which reduces this overhead by acquiring the memory access pattern of applications.

Because parallel applications often use the iterative methods, memory barriers are issued many times in a loop. Moreover, such applications usually access the shared memory with the fixed pattern. It means that if the memory access pattern and the communication set for maintaining consistency are determined at the first iteration, memory consistency may be realized by exchanging data according to the communication set.

FDSM inspects the access pattern of applications at a runtime using the memory management facility of the operating system. To achieve the single-byte granularity, the shared memory area is kept write protected and the page fault handler is invoked at each access to the area. The FDSM page fault handler records the accessed address and emulates the instruction being executed at the page fault so that the next instruction is executed after returning from the handler.

Omni/OpenMP[8] is the target OpenMP compiler for FDSM. It translates an OpenMP C/Fortran program to a C/Fortran program with a runtime system, a multi-thread runtime system for SMP or a software distributed shared memory system called SCASH.

FDSM has been implemented on top of the SCore cluster system software, which supports the user-level remote memory access facility using the Myrinet network. The CG benchmark program in the NAS parallel benchmarks is used to evaluate Omni OpenMP on FDSM. Since the current Omni OpenMP does not generate code for FDSM, the hand compiled code is used in this paper. The benchmark result shows that FDSM runs 40 % to 80 % faster than the SCASH DSM system and 70 % to 90 % slower than the CG program written in MPI.

2 Method

2.1 Overview

This section describes how FDSM provides the shared address space. FDSM makes use of the common characteristic of the numerical applications, that is, they often use the iterative methods and the accessed variables in a loop is fixed in each iteration. Therefore, FDSM analyzes the memory access pattern at the

first iteration of a loop and calculates the communication set. After the first iteration, FDSM uses this communication set to keep the coherence of the shared area.

2.2 Assumptions

FDSM may work as the normal DSM system which uses the twinning and diffing method[2]. With this method, there are no restrictions on the memory access pattern of the applications. However, FDSM may run faster if an application meets the following requirement: the access pattern to the shared memory is fixed in the loop.

This seems very strict, however many of the numerical calculations uses an iterative method, e.g. the Jacobi method, and such applications will meet this requirement.

2.3 Code Generation

This section describes how the OpenMP compiler generates the code for FDSM. We have not implemented the compiler yet, but it will be implemented soon by modifying the Omni/OpenMP compiler. This compiler may generate the source code for a DSM system called SCASH. As for the FDSM, the compiling scheme is almost same as SCASH. A little difference appears in the way to compile the "#pragma omp for" directive.

2.3.1 "#pragma omp for" directive

FDSM acquires the access pattern in a loop specified by the "#pragma omp for" directive. Function calls which tell the FDSM library to begin or finish acquiring the access pattern is inserted before and after the loop with this directive.

Figure 1 shows an example of the code generation. When function **fdsm_before_block(int id)** is called, FDSM begins acquiring the memory access pattern. The parameter *id* is a unique number to identify the corresponding loop statement. This number allows the run-time library to trace the flow of the program and detect the nested loop.

Function **fdsm_adjust_loop(int *, int *)** modifies both the initial and the last values of the loop counter to share the work inside the loop.

Function **fdsm_after_block()** tells the library to stop acquiring the access pattern and performs the barrier synchronization.

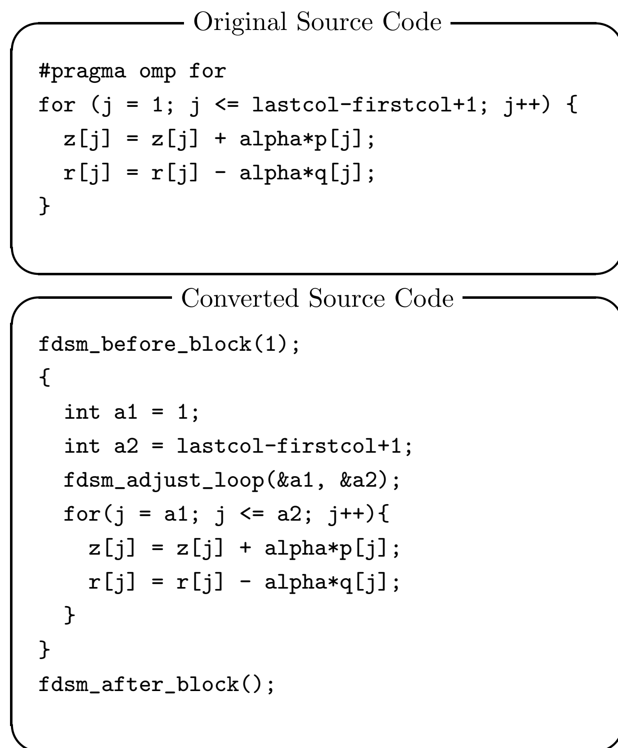

Figure 1: Converting the source code

2.3.2 Other directives

FDSM may support other directives such as *atomic* or *flush* by acting as the normal DSM system. So, the access pattern analysis is not used in such directives and the OpenMP compiler does nothing special.

2.4 Access Pattern Analysis

2.4.1 Granularity

Many of the software DSM system adopt the page size granularity. However, it is known that such DSM systems suffer from *false sharing*. In order to avoid this problem, FDSM adopts finer granularity.

FDSM obtains the write access pattern with the single-byte granularity. It is sufficient to avoid the false sharing problem because this fine-grain information allows FDSM to decide which variables have to be copied to other processors. On the other hand, read access pattern is obtained with the page-size granularity. This is because fine granularity is less important than that of the write access and getting too detailed access pattern will cause the performance

problems due to the page fault costs.

2.4.2 Access Pattern Inspection

The key point of FDSM is how to get the access pattern. We use the page protection mechanism of the memory management unit.

As we mentioned before, read access pattern is obtained with the page-size granularity. When FDSM begins the access pattern analysis, it makes all the shared pages read protected. If the application attempts to read these pages, the *page fault* will occur and the exception handler may record the accessed page. Then, the handler remove the read protection so that the next access to the same page should not cause a page fault.

The write access pattern is obtained with the single-byte granularity. As the read accesses, FDSM may use the *page fault* to record the accessed address. However, if FDSM makes the page write enabled, the next access to the same page will not cause the page fault and FDSM can't obtain the access pattern any more. On the other hand, if it does not make the page write enabled, the same instruction will cause the same exception immediately after returning from the handler.

To overcome this dilemma, the instruction emulation method is introduced. This method is described in the next section.

2.4.3 Instruction Emulation Method

The main idea of the instruction emulation method is that the exception handler writes the value on behalf of the instruction which caused the page fault. This method strongly depends on the architecture. We developed it for an Intel Pentium processor.

When a page fault occurs, the processor saves the current register values on the top of the kernel stack. When the execution is returning to the user code, the processor pops these values and restores them into the registers. It means that if the pushed value is changed, the processor restores the modified value.

The instruction emulation is realized using this mechanism. Instructions are emulated with the following five steps.

Step 1: Fetch Because the value of program counter is saved in the kernel stack, we access it and obtain the address of the instruction that caused the exception.

Step 2: Decode The kinds of an operation, operand registers, and the length of the instruction are decoded using the instruction address obtained in Step 1.

Step 3: Read If an operand register is an integer register, reading the operand is simply accessing the saved value. If an operand is in a floating point register, the value is not saved to the memory and still in the register. In this case, the value is copied from the register to a temporary memory area.

Step 4: Execute If the instruction being executed at the page fault is an arithmetic or logical operation, the instruction is emulated. If the instruction is only a copy operation, there is nothing to do in this step.

Step 5: Write FDSM may use the register value which represents the address where the write access has failed. By using this value, the destination address is known. The problem is that this region is write protected and copying the value to this address will fail again. However we have another way to access the same memory region.

In fact, the shared area of FDSM is maintained by the kernel and the user program maps this area. The page fault occurred because the user program maps this region with write protected. But the OS kernel maps the same area without any protection. So, the exception handler, which is the kernel mode program, may write to the page without any exceptions if it uses the kernel virtual address (Figure 2). In this way, operand write is completed.

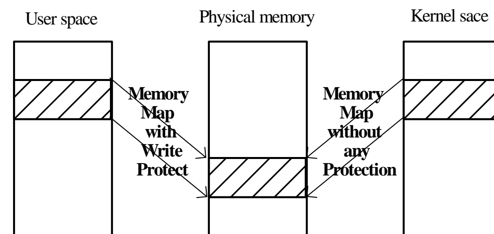

Figure 2: Operand Write

Now, the instruction emulation is finished and the value is written. At last, we have to increment the

program counter so that the processor should not re-execute the emulated instruction. The length of the instruction is known at the step 2. According to this length, the saved value of the program counter is incremented.

2.5 Communication Set Calculation

FDSM calculates the communication set for each block. If a variable is written in one block and the other processor reads it in other block, communication is required to update the variable. This section describes how to detect the variables which must be updated.

As an example, suppose an application whose execution flow is shown in Figure 3. In this figure, the boxes 1-4 represent the loops with the "#pragma omp for" directive. We call such a loop a *block*. Blocks 2 and 3 are inner loops of a looop called loop1. Loop 1 and block 4 are also inner loops of another loop. The access pattern in each block is known using the method described in the previous section.

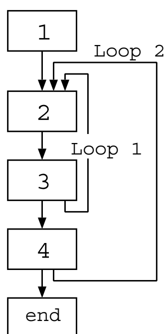

Figure 3: Execution Flow

The communication set is calculated for each block to be used in the communication after the block. In the example above, the variables to be updated after block 2 are such variables that are written in the block 2 and read by other processors in block 3 or 2 itself. FDSM searches such variables and adds them to the communication set. Note that the variables read in block 4 are not included in the communication set of block 2. If such variables are included, they are included in the communication every time block 2 is executed although they are not used during loop 1. They should be updated after loop 1, in other words, just before block 4 in order to eliminate the unnecessary communication.

FDSM treats the blocks inside a same loop as a single *parent block* when it calculates the communication set used after the loop of blocks. In the example, blocks 2 and 3 are treated as a single block (we call it block P1) and the communication set of P1 includes the variables written in block P1 and read in block 4.

In general, a loop with the "#pragma omp for" directive is a *block* and an outer loop of blocks is a *parent block*. Each block has its access pattern information. The access pattern of a *parent block* is obtained by merging those of child blocks. By using this access pattern information, FDSM inspects the dependency of variables between the bocks in the same level and make them the communication set.

To realize this, FDSM must detect the execution flow. Blocks made by the "#pragma omp for" directives may be detected by the OpenMP compiler. However, there are no OpenMP directive which is used as a clue to detect the nested loops. So, FDSM detects the execution flow at a run time using the *id* passed to the function **fdsm_before_block**. In the example in Figure 3, this function is called with the following id in this order: 1, 2, 3, 2, 3, 4, 2, 3, 2, 3, 4.

A flow of the execution is detected as shown in Figure 4. The meaning is as follows:

(1) Blocks 1, 2, and 3 are executed.

(2) Block 2 is executed again, and the run-time library founds that blocks 2 and 3 are in a loop and generates the parent block P1.

(3) Block 4 is executed and the tree grows.

(4) Block 2 is executed again. It means the blocks P1 and 4 are in a loop.

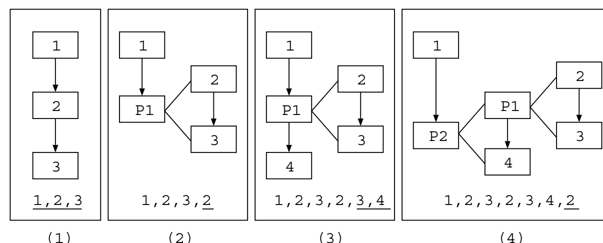

Figure 4: Loop Detection

2.6 Memory Barrier

By calculating the communication set, each process knows which variables have to be sent to other processes. Therefore, at the memory barrier point, FDSM

has only to copy such variables using remote memory write.

The actual procedure is as follows:

1. All the processes are synchronized to ensure no process accesses the shared memory.
2. Remote memory copies are performed.
3. All the processes are synchronized again to ensure the all data is updated.

2.7 Implementation Issues

FDSM consists of the kernel level driver and the user level library.

2.7.1 Kernel Level Driver

FDSM is implemented on top of the Linux operating system. The current implementation uses Linux 2.4.18.

The shared region is provided as the character device. User processes may access the region by mapping the device file using the mmap() system call. When the mmap() is called, the *vm area* for this region is created by the kernel. It contains a flag which represents the access protection mode of this region. FDSM modifies this flag to make the shared region read and write protected while the access pattern analysis. This flag is copied to the page table, which the processor refers at the memory access, when the physical page is actually allocated or when the protection mode is changed. FDSM needs not modify the page table directly.

The access pattern analyzer and the instruction emulator is implemented in the page fault handler. When a page fault occurs, the normal Linux kernel looks into the fault reason and performs the proper operations such as the *demand paging*, the *copy on write* or the operations for the illegal memory access. We modified this handler so that it determines at first whether the accessed area is in the FDSM area or the other normal memory region. If it is in the FDSM area, it calls the access pattern analyzer and the instruction emulator in FDSM. If it is not, the normal operation continues. Although most part of the FDSM kernel driver can be implemented as the kernel module, this modified handler requires the modification of the main kernel.

The access pattern information is passed to the user process by the memory mapped region. This region appears after the shared address space when the device file is mapped by a user process.

This driver also provides the functions of ioctl() system call. Some of them are used to initialize or finalize the FDSM driver. The most important one is used to change the protection mode of the shared region.

2.7.2 User Level Library

The user level library provides API's which may be used by the OpenMP compiler. Table 1 shows the list of API. Some of these API's issues system calls to use the functions of FDSM driver. For example, the initialization API maps the device file described above.

The most important role of the user level library is to find the communication set using the access pattern information obtained by the kernel level driver. This operation is performed when the memory barrier API is called and the communication set for this barrier has not yet been calculated. After the communication set is known, the library performs the remote memory write operations to keep the memory consistency.

FDSM runs on the SCore System Software[9]. FDSM uses the Myrinet network[1] and the PM/Myrinet[10] communication library as the communication layer, which provides the remote memory access facility in the user level.

Table 1: API

fdsm_init	Initializes the FDSM DSM system
fdsm_finalize	Finalizes the FDSM DSM system
fdsm_alloc	Allocates the shared memory area
fdsm_free	Releases the shared memory area
fdsm_before_loop	Begins the access pattern analysis
fdsm_after_loop	Stops the access pattern analysis and performs the barrier synchronization
fdsm_adjust_loop	Modifies the initial and the last values of the loop counter
fdsm_barrier	Performs the barrier synchronization (Used without the access pattern analysis)
fdsm_lock	Acquires the lock
fdsm_unlock	Releases the lock

3 Performance Evaluation

FDSM is evaluated using a 16 node Pentium III cluster. Table 2 shows the detailed specifications of this cluster.

Table 2: Specification of the PC cluster

# of nodes	16
CPU	Intel PentiumIII 500MHz
Cache	512KB
Chipset	440FX
Memory	512MBytes
Network	Myrinet 1.28GBits/sec
Node OS	Linux 2.4.18

CG in the NAS Parallel Benchmark is used as the benchmark application. The problem size is class A. The result is compared with the MPI and SCASH[3] versions of the same application. SCASH is a distributed shared memory system which adopts the twinning and diffing method and it also runs on SCore using the Myrinet network.

Note that three versions of CG are not the identical programs. Of course, communication code and the synchronizations are explicitly inserted in the MPI version of CG. SCASH uses the OpenMP version of CG compiled with the Omni/OpenMP compiler. FDSM uses the manually converted source code which is based on a compiled code of the Omni/OpenMP compiler.

3.1 CG result

Table 3 shows the results of CG and Figure 5 shows the speedup.

Table 3: CG

(Mops)

	1	2	4	8	16
FDSM	30.11	51.37	88.26	134.2	160.4
MPI	30.11	58.92	107.1	200.4	333.5
SCASH	30.11	54.14	80.67	95.48	84.84

FDSM shows the almost the linear scalability as long as the number of processors is no more than eight. However, it is saturated at this point. MPI version performs best and shows the linear scalability up to the sixteen nodes.

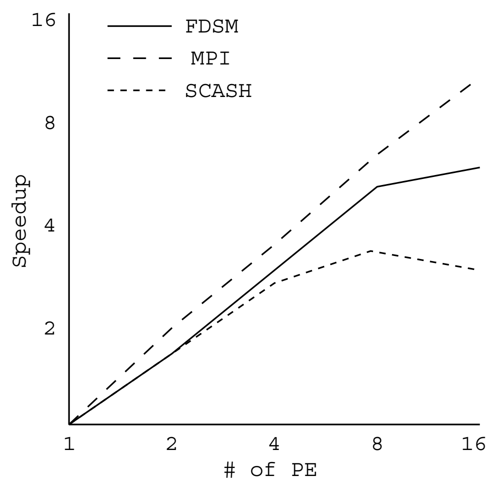

Figure 5: Speedup of CG

SCASH is scaled up up to four processors and then slows down with more processors.

This result shows the access pattern analysis may be better than the twinning and diffing method, employed by SCASH.

4 Discussion

FDSM does not perform well with sixteen processors. We think this is because FDSM analyzes the read access pattern with the coarse granularity and it causes the unnecessary communication. When a processor writes to a part of a page and other processor reads other part of that page, the communication should not occur. However, because FDSM does not know where is read in a page, this case cause the unnecessary communication. We think this is also the reason why MPI performs better than FDSM. MPI never performs the unnecessary communication.

If we want to reduce the unnecessary communication, FDSM has to get the read access pattern with the single byte granularity. This will cause the large overhead but the trade-off between this overhead and the cost of communication must be examined.

5 Related Work

The similar approach with FDSM is the inspector / executer method[7]. It uses the special compiler and divides the original computation into two phases: one is the inspector, which calculates the communication

pattern and another is the executer, which performs the actual computation. The difference is that FDSM does not use such a special compiler and it performs actual computation while it obtains the access pattern.

The producer-push method[4] is also similar in that it uses the execution history. It is the improvement of the twinning-diffing method. It predicts the next diff using the last recent used diff and send it before it is requested. The difference is that FDSM does not use the twin and diff. FDSM may detect the written area precisely even if the written value is same as the previous value.

There are several OpenMP compilers which generate the code for a DSM system. Omni OpenMP compiler[8] supports the SCASH, which is the DSM system using the twinning and diffing method. We will modify this compiler to generate the code for FDSM.

TreadMarks[5] is another software DSM and it has the OpenMP compiler[6]. This DSM system provides the special API's like fork and join for the OpenMP compiler.

6 Conclusion

This paper has described the new DSM system called FDSM. It reduces the overhead of the coherence maintenance by acquiring the access pattern of the applications.

The benchmark result has shown that FDSM performs 40% to 80% faster than SCASH. If the number of processor is less than eight, FDSM is scalable and its performance is 70% to 90% of the MPI version.

However if the number of processor is sixteen, the performance is about the half of the MPI version. We are inspecting the reason of this slowdown.

References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[3] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory. In *Proc. of HPC Asia 2000*, pages 158–163, 2000.

[4] Sven Karlsson and Mats Brorsson. Producer-push - a protocol enhancement to page- based software distributed shared memory systems. In *ICPP 1999*, 1999.

[5] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[6] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on network of workstations. In *Proc. of Supercomputing'98*, 1998.

[7] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[8] Mitsuhisa Sato, Hiroshi Harada, and Yutaka Ishikawa. OpenMP compiler for a software distributed shared memory system SCASH. In *WOMPAT2000*, July 2000.

[9] SCore. http://www.pccluster.org.

[10] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High performance communication middleware for heterogeneous network environments. 2000.