

OpenMP on Distributed Memory Computers with FDSM Distributed Shared Memory System

Hiroya Matsuba Yutaka Ishikawa
Graduate School of Information Science and Technology
The University of Tokyo
{matsuba, ishikawa}@is.s.u-tokyo.ac.jp

Abstract

FDSM, a software distributed shared memory system, has been designed and implemented to run OpenMP programs on distributed memory computers. FDSM inspects the access pattern of an application at the first iteration of a loop and figures out the communication pattern. FDSM is implemented on both IA-32 and IA-64 architectures and evaluated using the CG benchmark in the NAS parallel benchmarks. The result shows FDSM performs better than another SDSM system called SCASH.

1 Introduction

OpenMP programs run on the distributed memory parallel computers by using the software distributed shared memory (SDSM) system [5]. It is known that the performance of SDSM is limited because of the heavy overhead to maintain the coherence of the shared region.

FDSM [6] is the SDSM system for the applications that use loops. Many scientific applications use loops and some of those access the shared region with a fixed pattern. FDSM targets such applications. FDSM obtains the access pattern to the shared region at the first iteration of a loop. By using this access pattern, the communication pattern is determined and it is used after the second iteration. If the communication pattern is the same in every iteration, the overhead to determine the communication pattern is reduced.

FDSM assumes the Omni/OpenMP compiler environment. A code generated by the Omni/OpenMP compiler runs on FDSM.

We have proposed the method to inspect the access pattern on the IA-32 architecture in [6]. This paper proposes the new method to get the access pattern on the IA-64 architecture using its debug facilities.

FDSM is evaluated using the CG benchmark program in

the NAS Parallel Benchmarks on the IA-32 and IA-64 architectures. The result shows FDSM is faster than another software distributed shared memory system called SCASH on the IA-32 architecture. The result also shows FDSM performs almost same as the MPI version on the IA-64 architecture.

2 Target Applications

FDSM requires the applications to meet the following three conditions. The first condition is that the applications on FDSM must access the shared memory region with the fixed pattern. The second condition is that applications must explicitly declare the access to the shared memory region before they actually access the shared region if each process accesses the different variables by this access. The third condition is that applications must call the barrier operations when a memory barrier is required.

FDSM is designed to run the scientific applications on it. Scientific applications tend to use loops especially if they adopt iterative methods. Some of such applications access the shared memory region with the fixed pattern in each iteration of the loop, which meets the first condition.

It is the programmers' role to guarantee that the application meets the first condition. The second and third conditions are automatically satisfied by the Omni/OpenMP compiler as shown in the next section.

3 OpenMP programs on FDSM

3.1 Compilation

FDSM uses the Omni/OpenMP compiler to translate an OpenMP program to a normal C language source program. Omni/OpenMP compiler generates a source code for the SCASH [3] distributed shared memory system. When

```

/* The access pattern is fixed in the outer loop */
for (i = 0; i < IMAX; i++) {
  #pragma omp for
  for (j = 1; j <= JMAX; j++){
    z[j] = z[j] + alpha * p[j];
    r[j] = r[j] - alpha * q[j];
  }
}

```

Figure 1. OpenMP source

```

/* The access pattern is fixed in the outer loop */
for (i = 0; i < IMAX; i++) {
  {
    auto int j_41; auto int j_42; auto int j_43;
    (j_41)=(1);
    (j_42)=(JMAX+(1));
    (j_43)=(1);
    _ompc_default_sched(&j_41,&j_42,&j_43);
    for((j)=(j_41);(j)<(j_42);(j)+=(j_43)){
      ((*((z)+(j)))=((*((z)+(j)))+(>(*__G__L_alpha_4)*((*((p)+(j))))));
      ((*((r)+(j)))=((*((r)+(j)))-((*__G__L_alpha_4)*((*((q)+(j))))));
    }
  }
  _ompc_barrier();
} /* end i loop */

```

Figure 2. Converted OpenMP program

Omni converts an OpenMP program for SCASH, it relocates global variables to the shared region and parallelizes loops as indicated by the OpenMP directives. Figures 1 and 2 show an example of the source translation by the Omni/OpenMP compiler. Figure 1 is the original source program and Figure 2 is the translated code for SCASH.

As shown in Figure 2, the code generated by the Omni/OpenMP compiler does not directly call the SCASH API functions. For example, when the barrier operation is required, it calls **_ompc_barrier** instead of the SCASH API function. This means that the program generated by the Omni/OpenMP compiler runs on FDSM if the FDSM dedicated run-time library is linked. No modification of the Omni/OpenMP compiler is required.

Let us confirm that the source code shown in Figure 2 satisfies the conditions described in Section 2. Shared memory is accessed in the inner loop and the accessed region does not depend on the loop variable of the outer loop, i.e. the variable **i**. This means the access pattern is fixed in each iteration of the outer loop. So, the first condition is met. The second condition is satisfied because the function **_ompc_default_sched** indicates the beginning of the shared

access. Although the original role of this function is to schedule the execution of the loop to share the work among the processes, this function may be used to tell the FDSM run-time library that the shared access begins. The third condition is satisfied because the function **_ompc_barrier** is called to perform the barrier operation.

3.2 Overview of FDSM

This section describes how FDSM provides a virtual shared memory system. The basic idea of FDSM is shown in Figure 3. In the first iteration of a loop, FDSM obtains the access pattern of the applications and figures out the communication pattern as shown in Figure 3 (a). We call this execution the *inspection mode*.

In the rest of iterations, the communication is performed directly from the writers to the readers using the communication pattern obtained in the *inspection mode* as shown in Figure 3 (b). We call this type of execution the *fast execution mode*.

Let us define a term *access block* for the later discussion. The *access block* means a part of program where the each

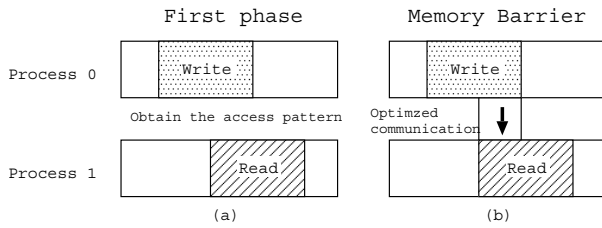


Figure 3. The idea of FDSM

process accesses to the assigned area of the shared region, i.e. the work is shared among the processes. In the example in Figure 2, the part between the `_ompc_default_sched` and `_ompc_barrier` forms a *access block*.

3.3 Inspection Mode

In the *inspection mode*, FDSM inspects the access pattern to the shared region. At the same time, FDSM provides the shared memory space with a home-based mechanism. In this mode, the application not only inspects the access pattern but also performs the normal calculations.

3.3.1 Access Pattern Inspection

In order to obtain the accurate communication pattern, the access pattern of the write access must be obtained with the 1-byte granularity. So, FDSM obtains the access pattern with this granularity. In order to reduce the read page faults, FDSM obtains the read access pattern with the page-size granularity. Unlike the write access, the 1-byte granularity is not necessary because even if FDSM judges a shared variable to be read but it is not actually read, this does not affect the correctness of the execution. We have already proposed the method to employ the page protection mechanism to get the access pattern in [6]. The overview of this method is as follows.

1. All the pages are made write protected.
2. When the application tends to write to the shared region, the page fault exception occurs.
3. The page fault handler records the accessed address and emulates the the instruction causing the page fault.
4. The execution is continued from the next instruction that caused the page fault.

Access Pattern Inspection on IA-64

On the IA-64 architecture, the access pattern may be obtained with more sophisticated way by using the facilities

to debug programs. Some modern architectures including IA64 provide provide **data break points**. This break point is set by the debugger on a memory address. The difference between a normal break point and a data break point is that a data break point is set on the memory address which contains data, not an instruction. When the application program loads from or stores to a memory region specified by the data break point, the debug exception occurs.

The IA-64 architecture provides the advanced data break point. On this architecture, the data break point may be not only a single address but also the continuous memory region. Using this debugging facility, FDSM obtains the access pattern in the following way:

1. The whole shared region is set to the data break point.
2. When the application accesses the shared region, the debug exception occurs.
3. The exception handler records the accessed address.

As shown above, FDSM needs not emulate the instruction that caused the debug exception. on the IA-64 architecture, while FDSM must emulate it on the IA-32 architecture. This difference comes from the difference of the kind of the exception. As for the page fault, the exception handler must remove the cause of the exception. After the execution of the exception handler finished, the instruction which caused the exception is re-executed and it should succeed. So, the page fault handler must remove the write protection of the pages on FDSM. But if FDSM does so, the future access to this page will not cause the exception and the access pattern cannot be obtained with the 1-byte granularity. This is because FDSM must emulate the instruction.

On the other hand, the debug facility provides the way to recover from the debug exceptions. In the debug exception rule, the instruction at the exception point is executed without more exception after returning from the debug exception handler. Because of this rule, FDSM needs not emulate the instruction.

3.3.2 Shared Memory Space in Inspection Mode

While FDSM inspects the access pattern of the application, it provides the shared memory region at the same time. This enables the application to perform the normal computation in the *inspection mode*.

Because all the access blocks are executed with the *inspection mode* when it is executed at the first time, FDSM knows the access pattern of all the previous executed access blocks. FDSM also records the time when a access block is executed. FDSM is able to compute the latest writer node of any shared area at any point of the execution. Figure 4 shows an example of this computation.

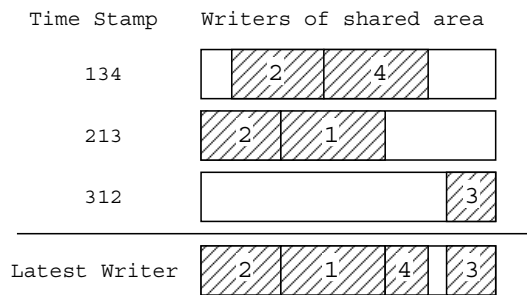


Figure 4. Calculation of the latest writer

At the beginning of the *inspection mode*, FDSM calculates the latest writer of each shared area and notifies the writer to send the latest value to the home node. Each shared area has one home node and it may be arbitrary chosen. In the current implementation of FDSM, the home node is determined using the round-robin fashion.

During the execution of an access block, the exceptions are raised to inspect the access pattern. While the exception handler records the accessed address, it copies the latest value from the home node. So, the user process correctly read the shared value.

3.4 Fast Execution Mode

In the *fast execution mode*, all the communication required to keep the coherence is performed at the beginning of each access block. FDSM knows the access pattern of all the previous executed access blocks and it also knows the access pattern of the next access block. So, FDSM is able to find out the required communication pattern so that the coming shared read may return the correct value. The communication pattern is decided as follows:

1. The latest writer of each shared area is computed.
2. The latest writer and the area which will be read in the next access block is compared. Communication is required in the area where its writer and reader is different.

Once the communication pattern is determined, this pattern may be used later when the same access block is executed again.

In the *fast execution mode*, no exception occurs during the execution of the user application. The overhead to maintain the memory consistency is only the time for communication.

3.5 Shared Access Outside of the Access Block

Applications may access the shared region outside of an access block. We define such a section as *semi access block*, which corresponds to a part of program outside of “omp for”. In the *semi access block*, as for the write access, all the processes concurrently write to the same address and it should be the same value. So, the memory consistency is automatically kept. As for the read access, all the previous written value must be read. So, FDSM obtains the read access pattern in the *inspection mode* and maintains the memory consistency in the same way as the normal *access block*.

4 Implementation Issues

4.1 IA-32

The IA-32 version of FDSM is implemented on the Linux kernel 2.4.21. This version of FDSM uses the page fault exception to obtain the access pattern in the *inspection mode*. Although the access pattern inspector may be implemented as the signal handler in the user space, we have implemented it in the kernel to reduce the overhead of the context switch. So, the page fault handler in the Linux kernel is modified. The instruction emulator is also implemented in the kernel.

The user level library of FDSM has two roles. One is to provide the application programming interfaces shown in Table 1. Another is to figure out the communication patterns in the *fast execution mode*.

FDSM runs on the SCore Cluster System Software[8]. PM/Myrinet[9] is used as the communication library.

Table 1. Provided API's

fdsm_init	Initializes FDSM
fdsm_finalize	Finalizes FDSM
fdsm_alloc	Allocates the shared region
fdsm_free	Frees the shared region
fdsm_before_loop	Tells FDSM the shared area will be accessed
fdsm_after_loop	Performs Barrier
fdsm_lock	Acquires a lock
fdsm_unlock	Releases a lock
fdsm_reduce	Performs a reduction operation

4.2 IA-64

The IA-64 version of FDSM is also implemented on Linux 2.4.21. This version of FDSM uses the debug facility

Table 2. Specification of the PC Cluster

	IA-32	IA-64
# of nodes	8	4
CPU	Xeon 2.8GHz	Itanium2 1.3GHz
Cache	512KB L2	3MB L2
Chipset	Intel E7501	
Memory	2Gbytes	2GB
Network	Myrinet 1.25Gbps Lanai 4.3 SAN cable	Myrinet 2.5Gbps Lanai 9.2 SAN cable

to get the access pattern. In order to use this facility, the debug registers must be correctly accessed. An IA-64 processor has at least 4 pairs of debug registers used to set the data break points. One pair consists of address and mask registers. Let the values of address and mask registers be a and m , respectively. When the memory access to the address R is issued, if $R \& m = a$, the debug exception occurs.

FDSM sets these debug registers using the ptrace system call. The access pattern inspector runs as a parent process and the user program runs as a child process. In the *inspection mode* when the user process accesses the shared region, the debug exception is reported to the parent process. The parent process records the accessed address and continue the execution of the child process.

The run-time library of the IA-64 version is same as that of the IA-32 version. The provided API's are also same as the IA-32 version and they are shown in table 1. As the IA-32 version, The IA-64 version of FDSM runs on the SCore Cluster System Software[8]. PM/Myrinet2k64 is used as the communication library.

5 Performance Evaluations

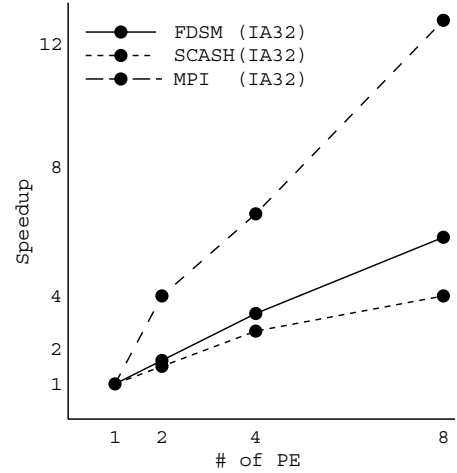
FDSM is evaluated using the CG program in NAS Parallel Benchmarks[1]. We used the OpenMP C language version of CG and the class is B. The OpenMP program is converted to a C program by the Omni/OpenMP compiler and linked with the FDSM library. The C compiler is gcc 3.2.2 on both the IA-32 and IA-64 architectures. This program runs on the 8-node IA-32 and the 4-node IA-64 clusters. The hardware environments are shown in Table 2.

Table 3 shows the performance. Figures 5 and 6 show the speedup on the IA-32 and IA-64 architectures, respectively. In these figures, the result of SCASH[3] (only IA-32) and the MPI is also shown as a comparison. SCASH is another software distributed shared memory system which runs on the SCore system software. OpenMP programs may run on SCASH using the Omni/OpenMP compiler.

On the IA-32 architecture, in the CG benchmark running

Table 3. CG

	1	2	4	8
FDSM(IA32)	68.38	128.5	234.7	389.9
SCASH(IA32)	68.38	114.7	195.7	281.7
MPI(IA32)	73.91	290.3	480.1	948.2
FDSM(IA64)	72.3	146.3	265.3	
MPI(IA64)	67.5	141.9	247.1	

**Figure 5. Speedup of the CG on IA-32**

with 2, 4, and 8 nodes, the execution is 1.88, 3.43, and 5.70 times faster than one node and FDSM is faster than SCASH by 1.12, 1.20, 1.38 times, respectively. The MPI version is much faster than the FDSM version. One reason is that FDSM has heavy overhead in the *inspection mode*. Another reason is that the MPI version of CG employs the block dimensional division to share the work among the processors while only the one-dimensional division is possible with OpenMP.

On the IA-64 architecture, the FDSM version performs almost same as MPI version. About the IA-64 version, this is the preliminary evaluation, so GCC is used as a backend compiler of the Omni/OpenMP compiler. More sophisticated compiler is required to exploit the performance of the IA-64 processor.

6 Related Work

The Inspector/Executor method[7] is similar to FDSM in that it obtains the access pattern of the applications. In this method, a special compiler divides a loop into two phases: the inspector and the executor. The access pattern of the

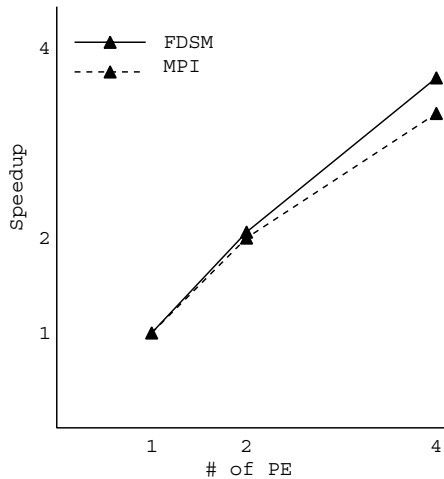


Figure 6. Speedup of the CG on IA-64

application is obtained by the inspector and the normal calculation is performed by the executor. Unlike the Inspector/Executor method, FDSM does not use the special compiler. While the inspector only gathers the access pattern of the application, the *inspection mode* of FDSM performs the normal calculations.

The producer-push method[4] is also similar to FDSM. It uses the execution history and predicts the next required communication. This method is the improvement of the twining and diffing method[2]. FDSM is different because it does not use twins nor diffs.

7 Conclusion

This paper has described the FDSM software distributed memory system. FDSM targets the application that uses loops. FDSM obtains the access pattern to the shared memory region at the first iteration of a loop. Using this access pattern, the communication is able to be optimized in the rest of the iterations.

FDSM has been implemented on the IA-32 and IA-64 architectures. This paper has described a method to implement FDSM on the IA-64 architecture. On this architecture, the access pattern of the application is obtained by using its advanced debugging facility.

For the evaluation of FDSM, the CG benchmark application has been used. The result shows FDSM performs better than SCASH by 38% on the IA-32 architecture. In the preliminary evaluation on the IA-64 architecture, FDSM performs almost same as the MPI version.

The current version of FDSM on IA-64 incurs heavy overhead because it is implemented as a user level library. FDSM on IA-64 may perform better if it is implemented in

the operating system kernel.

Acknowledgment

This research is partly supported by the Grant-in-Aid of Ministry of Education, Culture, Sports, Science and Technology in Japan, No. 14208026, 2002-2004.

References

- [1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
- [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] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.
- [6] Hiroya Matsuba and Yutaka Ishikawa. OpenMP on the FDSM software distributed shared memory. In *Proceedings of the Fifth European Workshop on OpenMP (EWOMP '03)*, pages 71 – 78, September 2003.
- [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] SCore. <http://www.pcluster.org>.
- [9] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High performance communication middleware for heterogeneous network environments. 2000.