

Performance Analysis and Improvement of OpenMP on Software Distributed Shared Memory Systems^{*}

Huang Chun and Yang Xuejun

National Laboratory for Parallel and Distributed Processing, P.R. China
{compiler}@sohu.com

Abstract. In this paper, the performance of the portable OpenMP compiler on SDSM JIAJIA is analyzed using SPEC OMPM2001 benchmark. The overheads of parallel execution have been investigated from the aspects of thread management and task schedule, memory access and synchronization. To improve the performance, the page placement and data privatization techniques have been implemented for the optimization of the compiler.

1 Introduction

OpenMP [1] is a portable, scalable model that provides shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from desktop to supercomputer. The OpenMP Application Program Interface (API) uses a directive-based programming paradigm to support incremental parallelization of an existing sequential program. Data communication and synchronization are managed by OpenMP compiler and runtime library. The programmer does not need to worry about the subtle details of the underlying architecture and operating system, such as thread management and the implementation of shared memory. Software distributed shared memory (SDSM) systems [2, 3] can provide a sufficiently practical emulation of shared memory across physically distributed memories, and are thus a possible candidate for basing an implementation of OpenMP upon. Implementing OpenMP on a SDSM system is one of the possible approaches to making OpenMP amenable to distributed memory systems, such as cluster architectures.

Some OpenMP compilers on SDSM are available. The OpenMP-NOW! Project aims to provide an OpenMP parallel programming environment on network of workstations [4]. It combines a source-to-source translator for OpenMP with the TreadMarks to enable OpenMP programs to execute on various platforms. The OpenMP directives and the default data attribute rules have been modified and extended to make the translator simpler. The Paramount group is implementing an OpenMP compiler based on Polaris compiler, and also uses TreadMarks

^{*} This work was supported by National 863 Hi-Tech Programme of China under Grant No. 2002AA1Z2105.

as the SDSM system [5, 6]. In these projects, the performance of OpenMP on SDSM systems has been measured, and it is found that the performance of real OpenMP applications is not so good as expected. However, the performance bottleneck has been analyzed only from program aspects [5].

We have completely implemented OpenMP 1.0 Fortran API on JIAJIA [2]. The tests of the small and highly parallel OpenMP programs show that one can get almost linear speedup on the SDSM system. Furthermore, in order to evaluate the compiler, we have measured the performance of SPEC OMPM2001 Fortran applications. It is discovered that the performance of these real OpenMP applications can hardly be improved when the number of processors is over 16. In this paper, the performance bottleneck of OpenMP applications is analyzed from three aspects beyond those inside programs, thread management and task schedule, memory access and synchronization. It is shown that memory access and synchronization are the main obstacles to improving performance. Then two techniques, page placement and data privatization, are implemented to optimize the compiler.

The rest of the paper is organized as follows. Section 2 briefly describes our portable compiler based on JIAJIA system. After measuring the performance of SPEC OMPM2001 Fortran applications, the bottleneck is carefully investigated in section 3. In section 4, two optimization techniques are proposed for improving performance of OpenMP program on SDSM systems. Finally, the paper is concluded with discussion.

2 The OpenMP Compiler

The OpenMP 1.0 Fortran API has been completely implemented on JIAJIA by our OpenMP compiler. To achieve portability, our compiler consists of two parts, a source-to-source translator and a runtime library. The objective of the source-to-source translator is just to translate OpenMP applications into normal Fortran programs with the runtime library calls. The library is the interface to JIAJIA and responsible for threads management and job schedule. When planning to port our compiler to others platforms, one can reuse the existing source-to-source translator and modify the library with the interface provided by those platforms.

2.1 JIAJIA SDSM System

JIAJIA [2] is the home-based SDSM system. It has two distinguished features compared to other SDSM systems such as TreadMarks. First, it combines physical memories of multiple nodes to form a larger shared space. Second, it provides data coherence with the scope consistency model [7], which is implemented through a locked-based protocol [8].

In JIAJIA, shared pages are allocated with system call `mmap`. Each shared page has a home node, and homes of shared pages are distributed across all nodes. The references to the remote pages cause these pages to be fetched from

their home and cached locally. The pages in the cache are in one of three states, `Invalid(INV)`, `Read-Only(RO)` and `Read-Write(RW)`.

2.2 Source-to-Source Translator

Our implementation is based on the fork-join programming model. First, all of threads are created at the beginning of execution and do some initialization operations, such as setting environment variables, allocating shared spaces, etc. Then only the master thread continues to execute serial region, and all the slave threads will be suspended once initialization is finished. Before entering a parallel region, the master wakes up all the slaves and broadcasts necessary information of the parallel region. The fork-join programming model is compatible to native OpenMP programming model and makes source-to-source translator simpler than the others based on SPMD model in which correctness must be ensured carefully.

The source-to-source translator encapsulates each parallel region into a separate subroutine. The task assignment and schedule are supported by the runtime library based on thread identifiers and schedule strategies specified by users.

2.3 Runtime Library

The runtime library is the bridge between OpenMP applications and JIAJIA system. It focuses on three tasks: thread management, job schedule, and implementation of OpenMP library routines and environment variables. The number of execution threads is decided by environment variable `OMP_NUM_THREADS` at the beginning of execution and always remains fixed over the duration of each parallel region. Nested parallel regions are serialized and dynamic adjustment of the number of execution threads is disabled. We have implemented four schedule strategies supported by OpenMP, i.e. static, dynamic, guided and runtime schedule.

3 Performance Measurement and Analysis

In this section we measure and analyze the performance of SPEC OMPM2001 benchmark on our cluster¹. The programs have been translated using our source-to-source translator and linked with the runtime library.

The SPEC OMPM2001 suite of benchmarks consists of real OpenMP C and Fortran applications [9, 10]. We only select Fortran benchmarks. Because the trends in the performance measurement are consistent for all nine Fortran applications. So, for brevity, in this paper, we primarily limit our analysis to the Swim benchmark, which is one of the most scalable benchmarks in SPEC OMPM2001.

¹ In the measurement we use a cluster architecture consisting of 8 alpha nodes. Each node contains one Compaq alpha 21264 processor and 1 Gbyte of memory. All the nodes are running OSF 4.0F and their page size is 8192 bytes. GNU G77 compiler is used.

3.1 Execution Time

Table 1 shows the execution time of Swim benchmark using the train dataset after 300 iterations. The speedup is acceptable in terms of user time from one to eight processors. However, considering total wall time, the overall speedup is much less. This fact is consistent with the nearly fixed system time from 2 to 8 processor execution. It is discovered that the system time is the result of shared-memory coherence activities.

Table 1. Execution Time for the Swim after 300 Iterations

number of threads	1	2	4	8
wall time	267.67	200.28	153.8	132.13
user time	267.48	165.51	114.8	96.45
system time	0.05	35.84	38.65	36.7

3.2 Performance Analysis

Swim code has eight parallel loops, with a reduction directive needed for one parallel loop. Its parallel coverage is about 99.5%, and the ideal speedup may be 7.7 running on 8 processors at most according to Amdahl's Law. But Table 1 shows that the overall speedup is only 2.03 on 8 processors. In the following, the performance bottleneck will be analyzed from the aspects of thread management and task schedule, memory access and synchronization.

Thread Management and Task Schedule All the threads are created at the beginning of execution and destroyed at the end of execution. The overhead of threads' creation can be omitted. Suspending and waking up the slaves are the main operations of thread management because of the fork-join programming model. All the slaves are suspended to wait to be waked up in serial regions. Before entering a parallel region, they know nothing about the task to be executed next time. The source-to-source translator encapsulates each parallel region into a separate subroutine, thus the master has to pass the start address of the subroutine and the information of all actual parameters to the slaves, besides it wakes up all the slaves. After receiving information, each of the slaves sets calling parameter at first, then jumps to the start address of the subroutine to execute the parallel region together with other slaves and the master. All information is packed in one structure. Because the communications based on shared variables usually exhibit significantly higher latencies, we use broadcast to pass the information necessary to execute the parallel region. Furthermore, broadcast is a blocked operation and can be used to suspend slaves. So, broadcast is used to suspend and wake up the slaves. Due to explicit message passing, the overhead of waking up slaves is quite small.

Swim code uses only static schedule. In static schedule, the task assignment and schedule are based on thread identification, the number of the execution threads and loop control variables. All the threads need not interact with each other, thus it only takes small time to partition and schedule task. But for dynamic and guided schedule, shared variables are used to control task assignment, which causes a mount of additional overhead.

Memory Access As many other SDSM systems, JIAJIA relies on the virtual memory page protection to detect writes. With this write detection scheme, the shared virtual memory pages are initially write-protected. System call `mprotect` is used for controlling how a page of memory may be accessed. If an access is disallowed by the given protection, a page fault will happen and the program will receive a `SIGSEGV` interrupt. The overhead of page fault T_{fault} is composed of two parts: overhead of entering and exiting from the interrupt service program T_{sig} and overhead of executing the interrupt service program T_{data} , i.e $T_{fault} = T_{sig} + T_{data}$. Page faults can be divided into local page faults and remote page faults. For local page faults, the main operation in the service program is to modify the page’s protection by `mprotect`. So $T_{fault} \approx T_{sig} + T_{modi}$. For remote page faults, T_{data} is more complex than that of a local page fault. It is mainly composed of two parts: protocol overhead $T_{protocol}$ and communication time T_{comm} . The protocol overhead contains T_{modi} , the time to update the cache for storing the page fetched from remote node and the additional time according to JIAJIA memory consistency protocol. T_{comm} will be discussed in the aspect of synchronization.

Most of the shared variables of Swim code are defined in the unnamed common block, which contains fourteen arrays, each of which is of $3802 \times 3802 \times 8$ bytes. 197638 shared pages are allocated only for these array variables. The number of local page faults plus remote page faults is always over 600,000 during the execution of Swim no matter how many processors are used. To measure the page fault overhead, we use the following simple program in Figure 1 to measure T_{sig} and T_{modi} .

Figure 2 shows that, the time of local page faults can not be ignored for large shared data, and half of the system time is caused by the virtual memory page protection to detect writes.

Synchronization There is an implicit barrier at beginning and end of a parallel region respectively. The number of iterations of the execution is 300 and Swim contains eight parallel regions, so the number of the barriers is 4800. As the development of relaxed memory consistency models [7] and the lazy implementation of cache coherence protocol [8], memory consistency activities will be almost always done at barrier points, so the overhead associated with the synchronization operation increases greatly. Synchronization overhead is composed of T_{comm} and the protocol overhead $T_{protocol}$. At barrier points, the states of all shared RW pages should be changed to RO by system call `mprotect`, which is an expensive operation as indicated by Figure 2. The execution time of the above

```

float *t;
void sigchild_handler(int sig, siginfo_t *sip, void *extra){
    mprotect(t,PAGESIZE,PROT_WRITE | PROT_READ);
}
main(){
    struct sigaction sa;
    t=(float*)malloc(PAGESIZE);
    t=(float*)((int)t+PAGESIZE-1)&~(PAGESIZE-1));
    sa.sa_sigaction = sigchild_handler;
    sigaction(SIGSEGV, &sa, 0);
    for(i=1;i<=600000;i++){
        mprotect(t,PAGESIZE,PROT_READ);
        *t = 0.3; // cause SIGSEGV
    }
}

```

Fig. 1. Program to Measure T_{sig} and T_{modi}

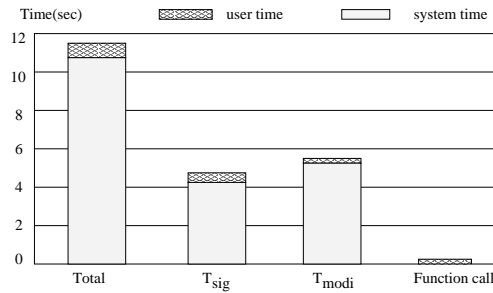


Fig. 2. Overhead of Page Faults ($T_{sig} + T_{modi}$)

operations is included $T_{protocol}$. On the other hand, T_{comm} is introduced by remote page faults and synchronization, and increases in proportion to the number of communications, which is quite large in a real application. It is found that communication bandwidth and latency are not the bottlenecks of performance, while the main obstacle is the number of communications. Therefore, optimization techniques must be used to reduce the number and size of communications for improving the performance, such as “owner computes”.

4 Optimization for Reducing Overheads

From the previous section, one can find that the system time is almost always caused by shared-memory coherence activities in SDSM systems. There are two kinds of approaches to improving performance of OpenMP.

Reduce remote memory access overhead such as page placement, data prefetch, dynamic page migration and schedule strategy.

Reduce shared data or shared pages such as data privatization and automatic data distribution.

We have implemented the two important optimization techniques in our compiler, i.e. page placement and data privatization.

4.1 Page Placement

The strategy for centralized data allocation and first-touch page placement is used in our compiler. The centralized data allocation contains two meanings. First, all shared variables of a program are allocated in the same place. In JIA-JIA, all the threads must participate in allocating shared variables. Furthermore, a shared variable must be allocated before its first use no matter the use appears in a serial region or a parallel region. It is quite complicated to wake up all the slaves to allocate shared variables together with the master in appropriate points during the execution. So a simpler method is used. All shared variables of a program are allocated after all the threads are created and before all the slaves are suspended for the first time.

Second, before the allocation, all non common block shared variables in a subroutine are collected into one block if they are not dummy arguments, then the block or a common block is allocated as a unit. The dummy arguments are excluded because they must be processed before the corresponding actual arguments are used. If the dummies are shared, it is required that the corresponding actual arguments are also shared. The call relationship between the dummies and the actual arguments must be kept. After the allocation, the address of a variable is computed according to the first address of the block and the size of its preceding variables which belong to the same allocation unit. For example, in Swim code, there are five allocation units, one for the unnamed common block, one for the common block `CONS` and the rest three for the blocks formed by the variables from three different subroutines, `SHALLOW`, `CALC1` and `CALC2`. So, the number of shared pages for the unnamed common block can be reduced from 197638 to 197630. Only one page is allocated for `CONS`, and three additional pages are allocated for the other allocation units. Twenty-six pages are allocated for these variables before the optimization.

The first-touch page placement is implemented by two steps. The first step makes all the pages of an allocation unit distribute across all the execution threads averagely because the allocation is done at the beginning of the execution. Here “threads” are used instead of “nodes”, which means if several threads are running on one node, the pages associated with these threads are all located in this node. If there are four threads to execute Swim, all the pages are allocated and distributed averagely as shown in Table 2.

The second step is to implement the first-touch placement based on home migration provided by JIAJIA [2]. If the page never migrated is referenced only by one thread in a parallel region, it will be migrated into the node on which

Table 2. Data allocation and page placement for Swim

thread identification	0	1	2	3
unnamed common block	49408	49408	49408	49406
CONS	1			
SHALLOW	1			
CALC1	1			
CALC2	1			

the thread is running. The timing is the implicit barrier point on exit from the parallel region. For a page which is ever migrated, its sequent migrations are controlled by a predefined threshold. The decrease of the number of shared pages makes the number of page faults decreased and improves the efficiency of cache greatly, as shown in Table 3.

Table 3. Execution Time for Swim after Page Placement

number of threads	1	2	4	8
wall time	248.27	172.54	112.32	84.01
user time	248.05	146.09	89.4	63.12
system time	0.05	24.05	22.74	20.89

Though the number of shared pages are reduced only 0.015% with the centralized allocation, the performance is improved greatly. One of the two reasons is to improve the efficiency of processor cache, which can be seen from serial time. Another is that the decrease of the number of shared pages makes the number of page faults decreased.

4.2 Data Privatization

In general, two kinds of shared data can be treated as private data [5, 6]. The shared data with read-only accesses in certain program sections can be made “private with copy-in” during these sections. Similarly, the shared data that are exclusively accessed by the same thread can be privatized during such a program phase.

The read-only shared variables are divided into arguments, common block variables, and non common block variables. Our compiler can identify the latter two kinds of variables and privatize them at present. All the non common block variables of a subroutine are allocated as a unit, so when a variable is collected, a particular access flag is used to mark how the variable is accessed in a parallel region. After the collection is finished, the variable can be privatized if its access flag is **R** which means it is read-only in all parallel regions of the subroutine. These variables are privatized by putting them into **FIRSTPRIVATE** clause.

An access flag is used for each common block. It is changed to W if any variable of the common block is modified in any parallel region of the program. In order to access the common block correctly, COPYIN clause must be used when necessary. So, the details must be recorded to get the information of the variable accesses in each parallel region. If the flag is already changed to W, the recording of the block will be stopped. The common block is privatized by changing its attributes to THREADPRIVATE and using COPYIN clause. In Swim code, all variables in the common block CONS are either read-only or never used in all parallel-region. Our compiler can assign CONS with THREADPRIVATE attribute and add COPYIN clause when necessary automatically.

```

!$OMP PARALLEL DO
  DO 100 J=1,N
  DO 100 I=1,M
    Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
1      -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
  50 CONTINUE

```

Fig. 3. Fragment from the first parallel region of CALC1

For example, the program in Figure 3 is a fragment from the first parallel region of subroutine CALC1. Variables N and M of CONS are used, and it is needed to add clause COPYIN(N, M) to broadcast their values. Variables FSDX and FSDY can be privatized with FIRSTPRIVATE clause. So, the directive becomes !\$OMP PARALLEL DO COPYIN(N, M) FIRSTPRIVATE (FSDX, FSDY).

Table 4 shows the execution time after the page placement and data privatization. The three pages referenced by all the threads frequently are eliminated, and the execution time is reduced.

Table 4. Execution Time for Swim after Page Placement and Data Privation

number of threads	1	2	4	8
wall time	247.67	169.28	107.9	77.13
user time	247.32	142.51	87.45	59.09
system time	0.05	22.75	20.05	17.79

5 Discussion

In this paper, we analyze the performance of our OpenMP compiler using the applications from SPEC OMPM2001 benchmark. The overheads of parallel ex-

ecution have been investigated carefully from the aspects of thread management and task schedule, memory access and synchronization. To improve the performance, the data privatization and page placement techniques have been implemented for the optimization of our compiler.

However, although the optimization techniques have improved the performance of OpenMP on SDSM systems greatly, it is far from satisfaction when running real applications. The system overhead, which is the bottleneck, is heavily caused by that the existing write detect mechanism of SDSM systems (e.g. JIAJIA) depends on the virtual memory page protection provided by operating systems. A promising approach is to build a hardware write-detection mechanism [11]. Therefore, the hardware, operating systems and compilers must cooperate in order to improve OpenMP program performance on SDSM systems.

References

1. The OpenMP Forum. OpenMP Fortran Application Program Interface, Version 1.0, October 1997 and OpenMP Fortran interpretations version 1.0, Apr. 1999. See <http://www.openmp.org>.
2. W. Hu, W. Shi, and Z. Tang. JIAJIA Software DSM System, Technical Report TR 980004, Inst. of Computing Technology, Chinese Academy of Sciences, 1998.
3. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18-28, February 1996.
4. H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on Networks of Workstations, in *Proc. of Supercomputing'98*, Nov. 1998.
5. A. Basumallik, S.-J. Min, and R. Eigenmann. Towards OpenMP execution on software distributed shared memory systems, *Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*, Lecture Notes in Computer Science 2327, Springer Verlag, May, 2002.
6. R. Eigenmann, J. Hoeflinger, R. Kuhn, D. Padua, A. Basumallik, S.-J. Min and J. Zhu, Is OpenMP for Grids? *Workshop on Next-Generation Systems, Int'l Parallel and Distributed Processing Symposium (IPDPS'02)*, May, 2002.
7. L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277-287, June 1996.
8. W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol, in *Proc. of the High Performance Computing and Networking (HPCN'99)*, LNCS 1593, pp. 463-472, Springer, Apr. 1999.
9. V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W.B. Jones, and B. Parady. SPECComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001)*, Lecture Notes in Computer Science, 2104, pages 1-10, July 2001.
10. H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, and M. van Waveren. Large System Performance of SPEC OMP2001 Benchmarks, *Lecture Notes in Computer Science 2327*, Springer Verlag, pp. 370-379, 2002.
11. N. Carter, W. Dally, W. Lee, S. Keckler and A. Chang. Processor Mechanisms for Software Shared Memory. In *International Symposium on High-Performance Computing*. Tokyo, Japan, Oct. 2000