Implementation and Evaluation of OpenMP for Hitachi SR8000

Yasunori Nishitani[1], Kiyoshi Negishi[1], Hiroshi Ohta[2], and Eiji Nunohiro[1]

[1] Software Development Division, Hitachi, Ltd.
Yokohama, Kanagawa 244-0801, Japan
{nisitani,negish_k,nunohiro}@soft.hitachi.co.jp
[2] Systems Development Laboratry, Hitachi, Ltd.
Kawasaki, Kanagawa 215-0013, Japan
ohta@sdl.hitachi.co.jp

Abstract. This paper describes the implementation and evaluation of the OpenMP compiler designed for the Hitachi SR8000 Super Technical Server. The compiler performs parallelization for the shared memory multiprocessors within a node of SR8000 using the synchronization mechanism of the hardware to perform high-speed parallel execution. To create an optimized code, the compiler can perform optimizations across inside and outside of a PARALLEL region or can produce a code optimized for a fixed number of processors according to the compile option. For user's convenience, it supports combination of OpenMP and automatic parallelization or Hitachi proprietary directive and also supports reporting diagnostic messages which help user's parallelization.
We evaluate our compiler by parallelizing NPB2.3-serial benchmark with OpenMP. The result shows 5.3 to 8.0 times speedup on 8 processors.

1 Introduction

Parallel programming is necessary to exploit high performance of recent supercomputers. Among the parallel programming models, the shared memory parallel programming model is widely accepted because of its easiness or incremental parallelization from serial programs. Until recently, however, to write a parallel program for shared memory systems, the user must use vendor-specific parallelization directives or libraries, which make it difficult to develop portable parallel programs.

To solve this problem, OpenMP[1][2] is proposed as a common interface for the shared memory parallel programming. The OpenMP Application Programming Interface(API) is a collection of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C,C++ programs.

Many computer hardware and software vendors are supporting OpenMP. Commercial and non-commercial compilers[3][4][5] are available on many platforms. OpenMP is also being used by Independent Software Vendors for its portability.

We implemented an OpenMP compiler for Hitachi SR8000 Super Technical Server. The SR8000 is a parallel computer system consisting of many nodes that incorporate high-performance RISC microprocessors and connected via a high-speed inter-node network. Each node consists of several Instruction Processor(IP)s which share a single address space. Parallel processing within the node is performed at high speed by the hardware mechanism called Co-Operative Microprocessors in a single Address Space(COMPAS).

Most implementations such as NanosCompiler[3] or Omni OpenMP compiler[4] use some thread libraries to implement the fundamental parallel execution model of OpenMP. As the objective of OpenMP for the SR8000 is to control the parallelism within the node and exploit maximum performance of IPs in the node, we implemented the fork-join parallel execution model of OpenMP over COMPAS, in which thread invocation is started by the hardware instruction, so that the overhead of thread starting or synchronization between threads can be reduced and high efficiency of parallel execution can be achieved.

The other characteristic of our OpenMP compiler is as follows.

– Can be combined with automatic parallelization or Hitachi proprietary directive
  Our OpenMP compiler supports full OpenMP1.0 specifications. In addition, our compiler can perform automatic parallelization or parallelization by Hitachi proprietary directives. Procedures parallelized by OpenMP can be combined with procedures parallelized by the automatic parallelization or by the Hitachi proprietary directives. This can be used to supplement OpenMP by Hitachi proprietary directives or to parallelize whole program by automatic parallelization and then to use OpenMP to tune an important part of the program.
– Parallelization support diagnostic messages
  Our compiler can detect loop carried dependence or recognize variables that should be given a PRIVATE or REDUCTION attribute and report the results of these analysis as diagnostic messages. The user can parallelize serial programs according to these diagnostic messages. It also is used to prevent incorrect parallelization by reporting warning messages if there is a possibility that the user's directive is wrong.
– Optimization across inside and outside of the PARALLEL region
  In OpenMP, the code block that should be executed in parallel is explicitly specified by the PARALLEL directive. Like many implementations, our compiler extracts PARALLEL region as a procedure to make implementation simple. Each thread performs parallel processing by executing that procedure. We designed our compiler that the extraction of PARALLEL region is done after global optimizations are executed. This enables optimizations across inside and outside of the PARALLEL region.
– Further optimized code generation by -procnum=8 option
  -procnum=8 option is Hitachi proprietary option with the purpose of bringing out the maximum performance from the node of SR8000. By default, our compiler generates an object that can run with any number of threads, but if -procnum=8 option is specified, the compiler generates codes especially optimized for the number of threads fixed to 8. This can exploit maximum performance of 8 IPs in the node.

In this paper, we describe the implementation and evaluation of our OpenMP compiler for the Hitachi SR8000. The rest of this paper is organized as follows. Section 2 describes an overview of the architecture of the Hitachi SR8000. Section 3 describes the structure and features of the OpenMP compiler. Section 4 describes the implementation of the OpenMP main directives. Section 5 describes the results of performance evaluation of the compiler. Section 5 also describes some problems about OpenMP specification found when evaluating the compiler. Section 6 concludes this paper.

2 Architecture of Hitachi SR8000

The Hitachi SR8000 system consists of computing units, called "nodes", each of which is equipped with multiple processors. Figure 1 shows an overview of the SR8000 system architecture.

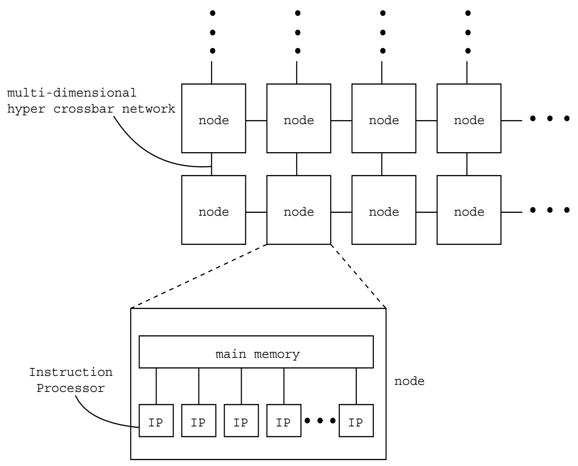

Fig. 1. Architecture of the SR8000 system

The whole system is a loosely coupled, distributed memory parallel processing system connected via high-speed multi-dimensional crossbar network. Each node has local memory and data are transferred via the network. The remote-DMA mechanism enables fast data transfer by directly transferring user memory space to another node without copying to the system buffer.

Each node consists of several Instruction Processor(IP)s. The IPs organize shared memory parallel processing system. The mechanism of Co-Operative Microprocessors in a single Address Space(COMPAS) enables fast parallel processing within the node.

The mechanism of COMPAS enables simultaneous and high-speed activation of multiple IPs in the node. Under COMPAS, one processor issues the "start instruction" and all the other processors start computation simultaneously. The "start instruction" is executed by hardware, resulting in high-speed processing. The operating system of SR8000 also has a scheduling mechanism to exploit maximum performance under COMPAS by binding each thread to a fixed IP.

As described above, SR8000 employs distributed memory parallel system among nodes and shared memory multiprocessors within a node that can achieve high scalability. The user can use message passing library such as MPI to control parallelism among nodes, and automatic parallelization or directive based parallelization of the compiler to exploit parallelism within the node. OpenMP is used to control parallelism within the node.

3 Overview of OpenMP Compiler

OpenMP is implemented as a part of a compiler which generates native codes for the SR8000. Figure 2 shows the structure of the compiler.

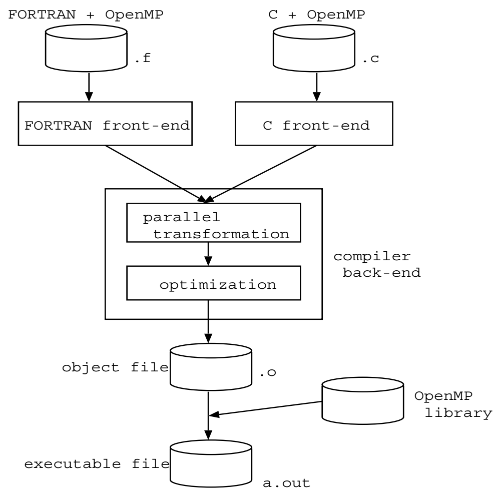

Fig. 2. Structure of the OpenMP compiler

The OpenMP API is specified for Fortran and C,C++. The front-end compiler for each language reads the OpenMP directives, converts to an intermediate language, and passes to the common back-end compiler. The front-end compiler for C is now under development.

The back-end compiler analyzes the OpenMP directives embedded in the intermediate language and performs parallel transformation. The principal transformation is categorized as follows.

- Encapsulate the PARALLEL region as a procedure to be executed in parallel, and generate codes that starts the threads to execute the parallel procedure.
- Convert the loop or block of statements so that the execution of the code is shared among the threads according to the work-sharing directives such as DO or SECTIONS.
- According to the PRIVATE or REDUCTION directives, allocate each variable to the thread private area or generate the parallel reduction codes.

– Perform necessary code transformation for the other synchronization directives.

The transformed code is generated as an object file. The object file is linked with the OpenMP runtime library and the executable file is created.

Our compiler supports full set of OpenMP Fortran API 1.0. All of the directives, libraries, and environment variables specified in the OpenMP Fortran API 1.0 can be used. However, nested parallelism and the dynamic thread adjustment are not implemented.

In addition, our compiler has the following features.

– Can be combined with automatic parallelization or Hitachi proprietary directives
  Procedures parallelized by OpenMP can be mixed together with procedures parallelized by automatic parallelization or Hitachi proprietary directives. By this feature, functions which do not exist in OpenMP can be supplemented by automatic parallelization or Hitachi proprietary directives. For example, array reduction is not supported in OpenMP 1.0. Then automatic parallelization or Hitachi proprietary directives can be used to parallelize the procedure which needs the array reduction and OpenMP can be used to parallelize the other procedures.
– Parallelization support diagnostic message
  Basically in OpenMP, the compiler performs parallelization exactly obeying the user's directive. However, when creating a parallel program from a serial program, it is not necessarily easy for the user to determine if a loop can be parallelized or if a variable needs privatization.
  For this reason, our compiler can perform the same analysis as the automatic parallelization even when OpenMP is used and report the result of the analysis as the diagnostic messages.

4 Implementation of OpenMP Directives

In this section, we describe the implementation of the main OpenMP directives.

4.1 PARALLEL Region Construct

In OpenMP, the code section to be executed in parallel is explicitly specified as a PARALLEL region. OpenMP uses the fork-join model of parallel execution. A program begins execution as a single process, called the master thread. When the master thread reaches the PARALLEL construct, it creates the team of threads. The codes in the PARALLEL region is executed in parallel and in a duplicated manner when explicit work-sharing is not specified. At the end of the PARALLEL region, the threads synchronize and only the master thread continues execution.

We implemented the PARALLEL region of OpenMP for SR8000 using COMPAS, so that the thread fork-join can be performed at high speed. Figure 3 represents the execution of PARALLEL region on COMPAS.

Under the COMPAS mechanism, each thread is bound to a fixed IP. The execution begins at the IP corresponding to the master thread, and the others wait for starting.

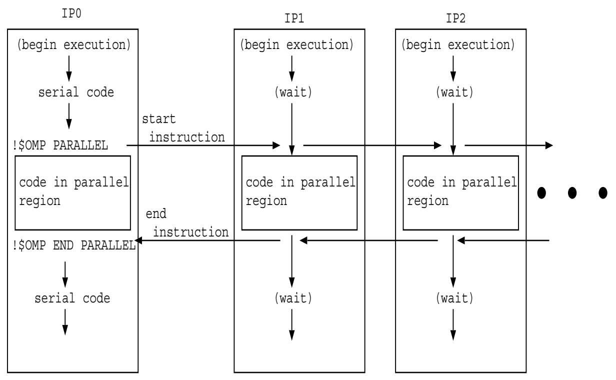

Fig. 3. Execution of PARALLEL region on COMPAS

When the master thread reaches the PARALLEL construct, the master IP issues the “start instruction” to the other IPs. This “start instruction” is executed by hardware resulting in high-speed processing. The IPs receiving the “start instruction” also receive the starting address of the PARALLEL region and immediately begin the execution simultaneously.

When PARALLEL region ends, each IP issues the “end instruction” and synchronizes. Then only the master IP continues the execution and the other IPs again enter the wait status.

Using COMPAS mechanism, the thread creation of OpenMP is performed by the hardware instruction to the already executing IPs, so that the overhead of thread creation can be reduced. Also as each thread is bound to a fixed IP, there is no overhead of thread scheduling.

The statements enclosed in PARALLEL region is extracted as a procedure inside the compilation process. The compiler generates a code that each thread runs in parallel by executing this parallel procedure. The PARALLEL region is extracted to a procedure to simplify the implementation. As the code semantics inside the PARALLEL region may differ from those outside the PARALLEL region, normal optimizations for serial programs cannot always be legal across the PARALLEL region boundary. Dividing the parallel part and non-parallel part by extracting the PARALLEL region allows normal optimizations without being concerned about this problem. The implementation of the storage class is also simplified by allocating the auto storage of the parallel procedure to the thread private area. The privatization of the variable in the PARALLEL region is done by internally declaring it as the local variable of the parallel procedure.

However, if the PARALLEL region is extracted as a procedure at the early stage of compilation, the problem arises that necessary optimization across inside and outside of the PARALLEL region is made unavailable. For example, if the PARALLEL region is extracted to a procedure before constant propagation, propagation from outside to inside of the PARALLEL region cannot be done, unless interprocedural optimization is performed.

To solve this problem, our compiler first executes basic, global optimization then extracts the PARALLEL region to a parallel procedure. This enables optimizations across inside and outside of the PARALLEL region. As described above, however, the same optimization as serial program may not be done as the same code in the parallel part and non-parallel part may have different semantics of execution. For example, the optimization which moves the definition of the PRIVATE variable from inside to outside the PARALLEL region should be prohibited. We avoid this problem by inserting the dummy references of the variables for which such optimization should be prohibited at the entry and exit point of the PARALLEL region.

4.2 DO Directive

The DO directive of OpenMP parallelizes the loop. The compiler translates the codes that set the range of loop index so that the loop iterations are partitioned among threads. In addition to parallelizing loops according to the user's directive, our compiler has the following features:

1. Optimization of loop index range calculation codes
   The code to set index range of parallel loop contains an expression which calculates the index range to be executed by each thread using the thread number as a parameter. When parallelizing the inner loop of a loop nest, the calculation of the loop index range will become an overhead if it occurs just before the inner loop. Our compiler performs the optimization that moves the calculation codes out of the loop nest if the original range of the inner loop index is invariant in the loop nest.
2. Performance improvement by -procnum=8 option
   In OpenMP, the number of threads to execute a PARALLEL region is determined by the environment variable or runtime library. This means the number of threads to execute a loop cannot be determined until runtime. So the calculation of the index range of a parallel loop contains the division expression of the loop length by the number of threads. While this feature increases the usability because the user can change the number of threads at runtime without recompiling, the calculation of the loop index range will become an overhead if the number of threads used is always the same constant value.
   Our compiler supports -procnum=8 option which aims at exploiting the maximum performance of the 8 IPs in the node. If -procnum=8 option is specified, the number of threads used in the calculation of the loop index range is assumed as the constant number 8. As the result, the performance is improved especially if the loop length is a constant as the loop length after parallelization is evaluated to a constant at the compile time and the division at runtime is removed.
3. Parallelizing support diagnostic message
   In OpenMP, the compiler performs parallelization according to the user's directive. It is the user's responsibility to ensure that the parallelized loop has no dependence across loop iterations or whether privatization is needed for each variable. However, it is not always easy to determine if a loop can be parallelized or to examine the needs of privatization for all the variables in the loop. Especially, once the user

```
1:       subroutine sub(a,c,n,m)
2:       real a(n,m)
3: !$OMP PARALLEL DO PRIVATE(tmp1)
4:       do j=2,m
5:         do i=1,n
6:           jm1=j-1
7:           tmp1=a(i,jm1)
8:           a(i,j)=tmp1/c
9:         enddo
10:      enddo
11:      end

(diagnosis for loop structure)
   KCHF2015K
             the do loop is parallelized by
             "omp do" directive. line=4
   KCHF2200K
             the parallelized do loop contains
             data dependencies across loop
             iterations. name=A line=4
   KCHF2201K
             the variable or array in do loop is
             not privatized. name=JM1 line=4
```

Fig. 4. Example of diagnostic message

gives an incorrect directive, it may take a long time to discover the mistake because of the difficulties of debugging peculiar to the parallel program.
For this reason, while parallelizing exactly obeying the user's directive, our compiler provides the function of reporting the diagnostic messages which is the result of the parallelization analysis of the compiler. The compiler inspects the statements in a loop and analyze whether there is any statement which prevents parallelization or variable that has dependence across loop iterations. It also recognizes the variables which need privatization or parallel reduction operation. Then if there is any possibility that the user's directive is wrong, it generates the dianostic messages. Figure 4 shows the example of the diagnostic message.
Also the compiler can report information for the loop with no OpenMP directives specified whether the loop can be parallelized or each variable needs privatization. This is useful when converting serial program to parallel OpenMP program.

4.3 SINGLE and MASTER Directives

SINGLE and MASTER directives both specify the statements enclosed in the directive to be executed once only by one thread. The difference is that SINGLE directive specifies the block of statements to be executed by any one of the threads in the team while MASTER specifies the block to be executed by the master thread. Barrier synchronization is inserted at the end of the SINGLE directive unless NOWAIT clause is specified, but no synchronization is done at the end of the MASTER directive and at the entry of the both directives.

The SINGLE directive is implemented that the first thread which reaches the construct executes the SINGLE block. This is accomplished by arranging SHARED attribute flag which means the block is already executed and accessed by each thread to determine whether the thread should execute the block or not. This enables timing adjustment if there is difference of execution timing among threads.

In contrast, the implementation of the MASTER directive is to add conditional branch that the block is executed only by the master thread.

The execution of the SINGLE directive is controled by the SHARED attribute flag, so the flag should be accessed exclusively by the lock operation. As the lock operation generally requires high cost of time, it becomes an overhead. As the result, in case that the execution timings of the threads are almost the same, using MASTER(+BARRIER) directive may show better performance than using SINGLE directive.

5 Performance

5.1 Measurement Methodology

We evaluated our OpenMP compiler by the NAS Parallel Benchmarks(NPB)[6]. The version used is NPB2.3-serial. We parallelized the benchmark by only inserting the OMP directives and without modifying the execution statement, though some rewriting is done where the program cannot be parallelized due to the limitation of current OpenMP specification(mentioned later). The problem size is class A.

The benchmark was run on one node of the Hitachi SR8000, and the serial execution time and parallel execution time on 8 processors are measured.

5.2 Performance Results

Figure 5 shows the result of performance.

The figure shows that the speedup on 8 processors is about 5.3 to 8.0 for 6 benchmarks except LU. The reason why the speedup of LU is low is that it contains wavefront style loop which every loop in the loop nest has dependence across loop iterations as follows, and the loop is not parallelized.

```
      do j=jst,jend
        do i=ist,iend
          do m=1,5
            v(m,i,j,k) = v(m,i,j,k)
            - omega*(ldy(m,1,i,j)*v(1,i,j-1,k)
     >              +ldx(m,1,i,j)*v(1,i-1,j,k)
     >              + ...
            ...
          enddo
         enddo
      enddo
```

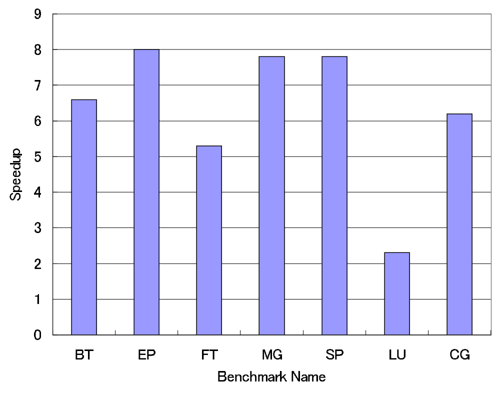

Fig. 5. Speedup of NPB2.3-serial

As the DO directive of OpenMP indicates that the loop has no dependence across loop iterations, this kind of loops cannot be parallelized easily.

The automatic parallelization of our compiler can parallelize such a wavefront style loop. We parallelized the subroutine with such a loop by automatic parallelization, parallelized the other subroutines by OpenMP, and measured the performance. The result on 8 processors shows the speedup of 6.3.

5.3 Some Problems of OpenMP Specification

When we parallelized the NPB or other programs by OpenMP, we met the case that the program cannot be parallelized or the program must be rewritten to enable parallelization, as some function do not exist in OpenMP. We describe some of the cases below.

1. Parallelization of a wavefront style loop
   As mentioned in section 5.2, we met the problem that a wavefront style loop cannot be parallelized in the LU benchmark of NPB. This was because OpenMP has only directives which mean that the loop has no dependence across loop iterations.
2. Array reduction
   In the EP benchmark of NPB, parallel reduction for an array is needed. However, in the OpenMP 1.0 specification, only the scalar variable can be specified as REDUCTION variable and arrays cannot be specified. This causes rewriting of the source code.
3. Parallelization of loop containing induction variable
   It is often needed to parallelize the loop that involves an induction variable as follows.

```
K=...
do I=1,N
  A(2*I-1)=K
```

```
      A(2*I)=K+3
      K=K+6
    enddo
```

However, also in this case it cannot be parallelized only with the OpenMP directive and needs to modify the source code.

As these kind of parallelizations described above often appear when parallelizing real programs, it is desirable to extend OpenMP so that these loops can be parallelized only by OpenMP directives.

6 Conclusions

In this paper, we described the implementation and evaluation of the OpenMP compiler for parallelization within the node of Hitachi SR8000. This compiler implements the fork-join execution model of OpenMP using hardware mechanism of SR8000 and achieves high efficiency of parallel execution. We also made our compiler possible to perform optimization across inside and outside of the PARALLEL region or to generate codes optimized for 8 processor by -procnum=8 option. Furthermore, for user's convenience, we implemented parallelizing support diagnostic messages that help the user's parallelization, or enabled combination with automatic parallelization or Hitachi proprietary directives. We evaluated this compiler by parallelizing NAS Parallel Benchmarks with OpenMP and achieved about 5.3 to 8.0 speedup on 8 processors.

Through this evaluation of OpenMP, we found several loops cannot be parallelized because there are features which OpenMP lacks. These loops can be parallelized by automatic parallelization or Hitachi proprietary directives provided by our compiler. We intend to develop the extension to the OpenMP specification so that these loops can be parallelized only by directives.

References

[1] OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. *White paper*, Oct 1997.

[2] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Oct 1997 1.0.

[3] Eduard Ayguadé, Marc Gonzàlez, Jesús Labarta, Xavier Martorell, Nacho Navarro, and José Oliver. NanosCompiler. A Research Platform for OpenMP extensions. In *EWOMP'99*.

[4] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *EWOMP'99*.

[5] Christian Brunschen and Mats Brorsson. OdinMP/CCp - A portable implementation of OpenMP for C. In *EWOMP'99*.

[6] NASA. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

Copyright Information