Performance Evaluation of the Omni OpenMP Compiler

Kazuhiro Kusano, Shigehisa Satoh and Mitsuhisa Sato

RWCP Tsukuba Research Center, Real World Computing Partnership
1-6-1, Takezono, Tsukuba-shi, Ibaraki, 305-0032, JAPAN
TEL: +81-298-53-1683, FAX: +81-298-53-1652
e-mail: {kusano,sh-sato,msato}@trc.rwcp.or.jp
http://pdplab.trc.rwcp.or.jp/Omni/

Abstract. We developed an OpenMP compiler, called Omni. This paper describes a performance evaluation of the Omni OpenMP compiler. We take two commercial OpenMP C compilers, the KAI GuideC and the PGI C compiler, for comparison. Microbenchmarks and a program in Parkbench are used for the evaluation. The results using a SUN Enterprise 450 with four processors show the performance of Omni is comparable to a commercial OpenMP compiler, KAI GuideC. The parallelization using OpenMP directives is effective and scales well if the loop contains enough operations, according to the results.

keywords: *OpenMP, compiler, Microbenchmarks, parkbench, performance evaluation*

1 Introduction

Multi-processor workstations and PCs are getting popular, and are being used as parallel computing platforms in various types of applications. Since porting applications to parallel computing platforms is still a challenging and time consuming task, it would be ideal if it could be automated by using some parallelizing compilers and tools. However, automatic parallelization is still a challenging research topic and is not yet at the stage where it can be put to practical use.

OpenMP[1], which is a collection of compiler directives, library routines, and environment variables, is proposed as a standard interface to parallelize sequential programs. The OpenMP language specification came out in 1997 for Fortran, and in 1998 for C/C++. Recently, compiler vendors for PCs and workstations have endorsed the OpenMP API and have released commercial compilers that are able to compile an OpenMP parallel program.

There have been several efforts to make a standard for compiler directives, such as OpenMP and HPF[12]. OpenMP aims to provide portable compiler directives for shared memory programming. On the other hand, HPF was designed to provide data parallel programming for distributed or non-uniform memory access systems. These specifications were originally supported only in Fortran, but OpenMP announced specifications for C and C++. In OpenMP and HPF, the directives specify parallel actions explicitly rather than as hints for parallelization.

While high performance computing programs, especially for scientific computing, are often written in Fortran as the programming language, many programs are written in C in workstation environments. We focus on OpenMP C compilers in this paper. We also report our evaluation of the Omni OpenMP compiler[4] and make a comparison between Omni and commercial OpenMP C compilers. The objectives of our experiment are to evaluate available OpenMP compilers including our Omni OpenMP compiler, and examine the performance improvement gained by using the OpenMP programming model.

The remainder of this paper is organized as follows: Section 2 presents the overview of the Omni OpenMP compiler and its components. The platforms and the compilers we tested for our experiment are described in section 3. Section 4 introduces Microbenchmarks, an OpenMP benchmark program developed at the University of Edinburgh, and shows the results of an evaluation using it. Section 5 presents a further evaluation using another benchmark program, Parkbench. Section 6 describes related work and we conclude in section 7.

2 The Omni OpenMP Compiler

We are developing an experimental OpenMP compiler, Omni[4] , for an SMP machine. An overview of the Omni OpenMP compiler is presented in this section.

The Omni OpenMP compiler is a translator which takes OpenMP programs as input and generates multi-thread C programs with run-time library calls. The resulting programs are compiled by a native C compiler, and then linked with the Omni run-time library to execute in parallel. The Omni is supported the POSIX thread library for parallel execution, and this makes it easy to port the Omni to other platforms. The platforms the Omni has already been ported to are the Solaris on Sparc and on intel, Linux on intel, IRIX and AIX.

The Omni OpenMP compiler consists of three parts, a front-end, the Exc Java tool and a run-time library. Figure 1 illustrates the structure of Omni.

The Omni front-end accepts programs parallelized using OpenMP directives that are specified in the OpenMP application program interface[2][3]. The front-end for C and FORTRAN77 are available now, and a C++ version is under development. The input program is parsed into an Omni intermediate code, called Xobject code, for both C and FORTRAN77.

The next part, the Exc Java tool, is a Java class library that provides classes and methods to analyze and transform the Xobject intermediate code. It also generates a parallelized C program from the Xobject. The representation of Xobject code which is manipulated by the Exc Java tool is a kind of Abstract Syntax Tree(AST) with data type information. Each node of the AST is a Java object that represents a syntactical element of the source code that can be easily transformed. The Exc Java tool encapsulates the parallel execution part into a separate function to translate a sequential program with OpenMP directives into a fork-join parallel program.

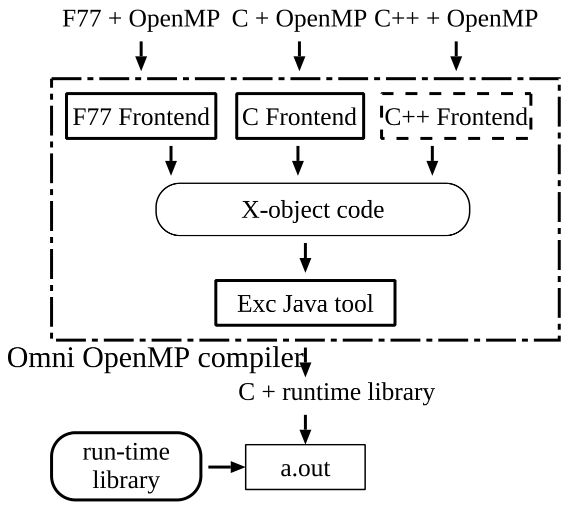


Fig. 1. Omni OpenMP compiler

Figures 2 and 3 show the input OpenMP code fragment and the parallelized code which is translated by Omni, respectively. A master thread calls the Omni

```
func(){
  ...
#pragma omp parallel for
  for(...){
    x=y...
  }
}
```

Fig. 2. OpenMP program fragment

run-time library, _ompc_do_parallel, to invoke slave threads which execute the function in parallel. Pointers to shared variables with auto storage classes are copied into a shared memory heap and passed to slaves at the fork. Private variables are redeclared in the functions generated by the compiler. The work sharing and synchronization constructs are translated into codes which contain the corresponding run-time library calls.

The Omni run-time library contains library functions used in the translated program, for example, _ompc_do_parallel in Figure 3, and libraries that are specified in the OpenMP API. For parallel execution, the POSIX thread library and

```
void __ompc_func_6(void **__ompc_args)
{
  auto double **_pp_x;
  auto double **_pp_y;
  _pp_x = (double **)*(__ompc_args+0);
  _pp_y = (double **)*(__ompc_args+1);
  {
   /* index calculation */
    for(...){
      __p_x=__p_y...
    }
  }
}
func(){
  ...
  {/* #pragma omp parallel for */
    auto void *__ompc_argv[2];
    *(__ompc_argv+0) = (void *)&x;
    *(__ompc_argv+1) = (void *)&y;
    _ompc_do_parallel(__ompc_func_6,__ompc_argv);
  }
```

Fig. 3. Program parallelized using Omni

the Solaris thread library on Solaris OS can be used according to the Omni compilation option. The Omni compilation option also allows use of the mutex_lock function instead of the spin-wait lock we developed, the default lock function in Omni. The 1-read/n-write busy-wait algorithm[13] is used as a default Omni barrier function.

Threads are allocated at the beginning of an application program in Omni, not at every parallel execution part contained in the program. All threads but the master are waiting in a conditional wait state until the start of parallel execution, triggered by the library call described before. The allocation and deallocation of these threads are managed by using a free list in the run-time library. The list operations are executed exclusively using the system lock function.

3 Platforms and OpenMP Compilers

The following machines were used as platforms for our experiment.

- SUN Enterprise 450(Ultra sparc 300MHz x4), Solaris 2.6, SUNWspro 4.2 C compiler, JDK1.2
- COMPaS-II(COMPAQ ProLiant6500, Pentium-II Xeon 450MHz x4), RedHat Linux 6.0+kernel-2.2.12, gcc-2.91.66, JDK1.1.7

We evaluated commercial OpenMP C compilers as well as the Omni OpenMP compiler. The commercial OpenMP C compilers we tested are:

- KAI GuideC Ver.3.8[10] on the SUN, and
- PGI C compiler pgcc 3.1-2[11] on the COMPaS-II.

KAI GuideC is a preprocessor that translates OpenMP programs into parallelized C programs with library calls. On the other hand, the PGI C compiler translates an input program directly to the executable code. The compile options used in the following tests are '-fast' for the SUN C compiler, '-O3 -malign-double' for the GNU gcc, and '-mp -fast' for the PGI C compiler.

4 Performance Overhead of OpenMP

This section presents the evaluation of the performance overhead of OpenMP compilers using Microbenchmarks.

4.1 *Microbenchmarks*

Microbenchmarks[6], developed at the University of Edinburgh, is intended to measure the overheads of synchronization and loop scheduling in the OpenMP runtime library. The benchmark measures the performance overhead incurred by the OpenMP directives, for example 'parallel', 'for' and 'barrier', and the overheads of the parallel loop using different scheduling options and chunk sizes.

4.2 Results on the SUN System

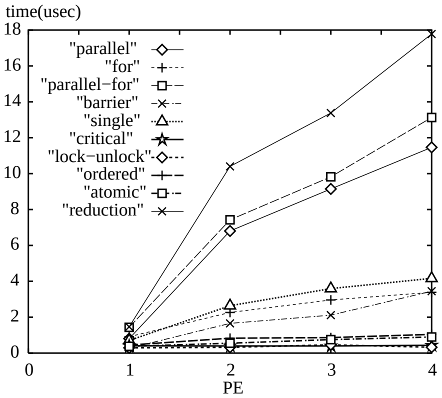
Figure 4 shows the results of using the Omni OpenMP compiler and KAI GuideC. The native C compiler used for both OpenMP compilers is the SUNWspro 4.2 C compiler with the '-fast' optimization option.

These results show the Omni OpenMP compiler achieves competitive performance when compared to the commercial KAI GuideC OpenMP compiler. The overhead of 'parallel', 'parallel-for' and 'parallel-reduction' is bigger than that of other directives. This indicates that it is important to reduce the number of parallel regions to achieve good parallel performance.

4.3 Results on the COMPaS-II System

The results of using the Omni OpenMP compiler and the PGI C compiler on the COMPaS-II are shown in Figure 5.

The PGI compiler shows very good performance, especially for 'parallel', 'parallel-for' and 'parallel-reduction.' The overhead of Omni for those directives increases almost linearly. Although the overhead of Omni for those directives is twice that of PGI, it is reasonable when compared to the results on the SUN.



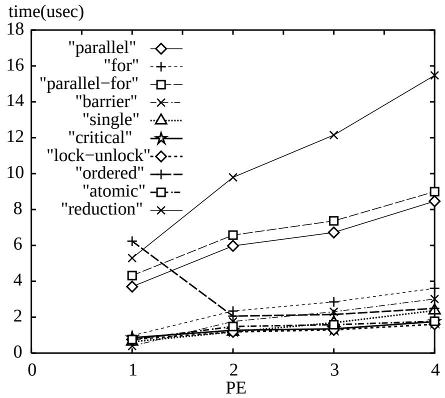
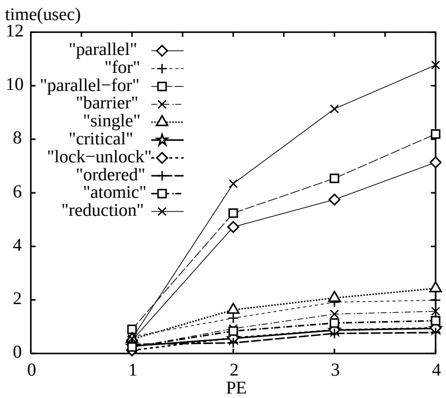


Fig. 4. Overhead of Omni(left) and KAI(right)



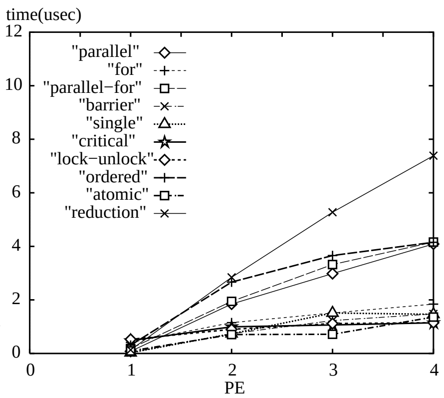


Fig. 5. Overhead of Omni(left) and PGI(right)

4.4 Breakdown of the Omni Overhead

The performance of 'parallel', 'parallel-for' and 'parallel-reduction' directives originally scales poorly on Omni. We made some experiments to breakdown the overhead of the 'parallel' directive, and, as a result, we found that the data structure operation used to manage parallel execution and synchronization in the Omni run-time library spent most of the overhead.

The threads are allocated once the initialization phase of a program execution, and, after that, idle threads are managed by the run-time library using an idle queue. This queue has to be operated exclusively and this serialized queue operations. In addition to the queue operation, there is a redundant barrier synchronization at the end of the parallel region in the library. We modified the run-time library to reduce the number of library calls which require exclusive operation and eliminate redundant synchronization. As a result, the performance shown in Figures 4 and 5 are achieved. Though the overhead of 'parallel for' on the COMPaS-II is unreasonably big, the cause of this is not yet fixed.

Table 1 is the time spent for an allocation of threads and a release of threads and barrier synchronization on the COMPaS-II system. This shows thread allocation still spent the most of the overhead.

PE	1	2	3	4
allocation	0.40(43)	2.7(67)	3.5(65)	4.0(63)
release + barrier	0.29(31)	0.50(12)	0.56(10)	0.60(9)

Table 1. Time to allocate/release data(usec(%))

5 Performance Improvement from using OpenMP Directives

This section describes the performance improvements using the OpenMP directives.

We take a benchmark program from Parkbench to use in our evaluation. The performance improvements of a few simple loops with the iterations ranging from one to 100,000 show the efficiency of the OpenMP programming model.

5.1 *Parkbench*

Parkbench[8] is a set of benchmark programs designed to measure the performance of parallel machines. Its parallel execution model is message passing using PVM or MPI. It consists of low-level benchmarks, kernel benchmarks, compact applications and HPF benchmarks.

We use one of the programs, rinf1, in the low-level benchmarks to carry out our experiment. The low-level benchmark programs are intend to measure the performance of a single processor. We rewrote the rinf1 program in C, because the original was written in Fortran. The rinf1 program takes a set of common Fortran operation loops in different loop lengths. For the following test, we chose kernel loops 3, 6 and 16. Figure 6 shows code fragments from a rinf1 program.

5.2 Results on the SUN System

Figures 7, 8 and 9 show the results of kernel loops 3, 6 and 16, respectively, in the rinf1 benchmark program which was parallelized using OpenMP directives

```
 for( jt = 0 ; jt < ntim ; jt++ ){
   dummy(jt);
#pragma omp parallel for
   for( i = 0 ; i < n ; i++ )/* kernel 3 */
     a[i] = b[i] * c[i] + d[i];
 }
 ....
#pragma omp parallel for
   for( i = 0 ; i < n ; i++ )/* kernel 6 */
     a[i] = b[i] * c[i] + d[i] * e[i] + f[i];
...
```

Fig. 6. rinf1 kernel loop

executed on the SUN machine. In these graphs, the x-axis is loop length, and the y-axis represents performance in Mflops.

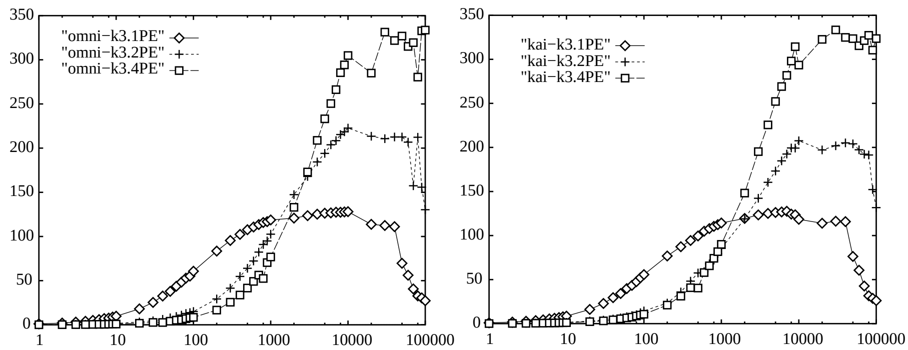


Fig. 7. kernel 3[a(i)=b(i)*c(i)+d(i)] on the SUN: Omni(L) and KAI(R)

Both OpenMP compilers, Omni and KAI GuideC, achieve almost the same performance improvement, though there are some differences. The differences resulted mainly from the run-time library, because both OpenMP compilers translate to the C program with run-time library calls. KAI GuideC shows better performance for short loop lengths of kernel 6 on one processor, and the peak performance for kernel 16 on two and four processors is better than that of Omni.

5.3 Results on the COMPaS-II System

Figures 10, 11 and 12 are the results of kernel loops in the rinf1 benchmark program which were parallelized using the OpenMP directive executed on the

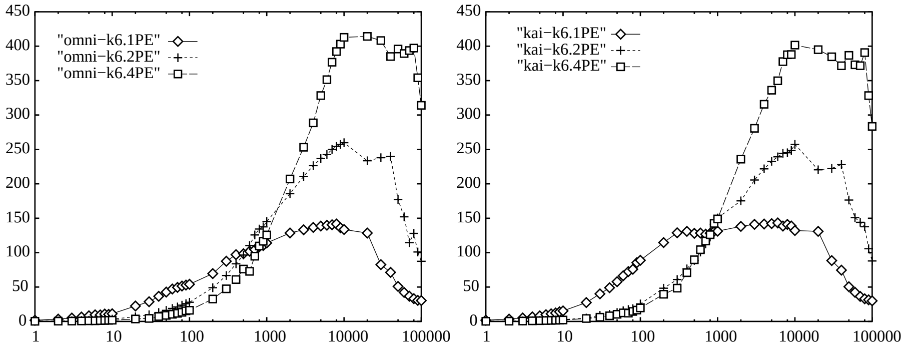


Fig. 8. kernel 6[a(i)=b(i)*c(i)+d(i)*e(i)+f(i)] on the SUN: Omni(L) and KAI(R)

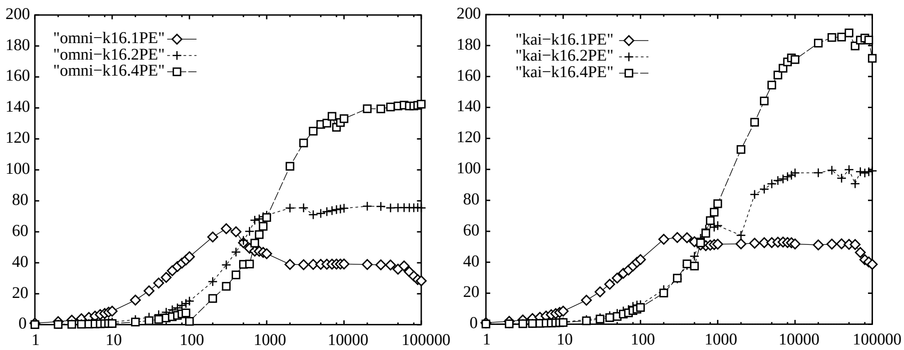


Fig. 9. kernel 16[a(i)=s*b(i)+c(i)] on the SUN: Omni(L) and KAI(R)

COMPaS-II. The x-axis represents loop length, and the y-axis represents performance in Mflops, the same as in the previous case.

The results show the PGI compiler achieves better performance than the Omni OpenMP compiler on the COMPaS-II. The PGI compiler achieves very good performance for short loop lengths on one processor. The peak performance of PGI reaches about 400 Mflops or more on four processors, and it is nearly double that of Omni in kernels 3 and 16.

5.4 Discussion

Omni and KAI GuideC achieve almost the same performance improvement on the SUN, but the points described above must be kept in mind. The performance improvement of the PGI compiler on the COMPaS-II has different characteristics when compared to the others. Especially, the PGI achieves higher performance

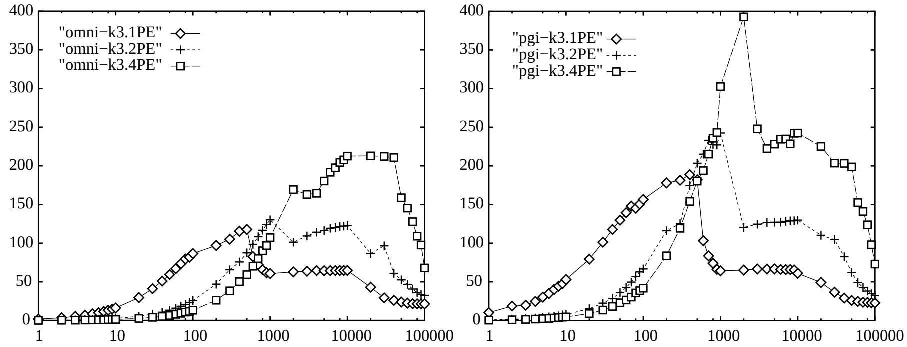


Fig. 10. kernel 3[a(i)=b(i)*c(i)+d(i)] on the COMPaS-II: Omni(L) and PGI(R)

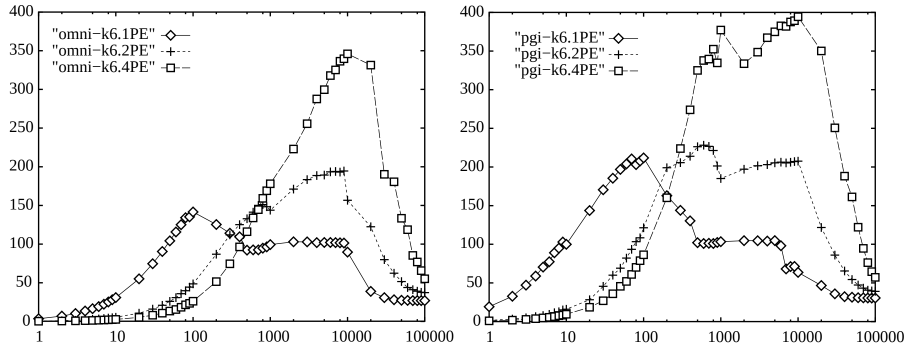


Fig. 11. kernel 6[a(i)=b(i)*c(i)+d(i)*e(i)+f(i)] on the COMPaS-II: Omni(L) and PGI(R)

for short loop lengths than the Omni on one processor, and the peak performance nearly doubles for kernel 3 and 16. This indicates the performance of Omni could be improved on the COMPaS-II by the optimization of the Omni runtime library, though one must consider the fact that the backend of Omni is different.

Those results show that parallelization using the OpenMP directives is effective and the performance scales up for tiny loops if the loop length is long enough.

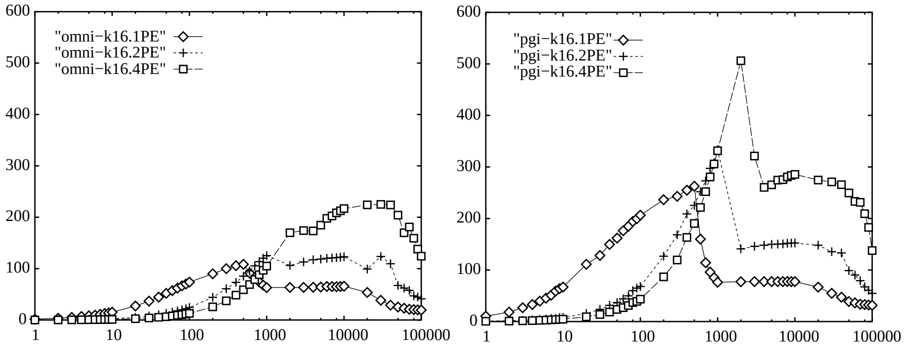


Fig. 12. kernel 16[a(i)=s*b(i)+c(i)] on the COMPaS-II: Omni(L) and PGI(R)

6 Related Work

Lund University in Sweden developed a free OpenMP C compiler, called OdinMP/CCp[5]. It is also a translator to a multi-thread C program and uses Java as its development language, the same as our Omni. The difference is found in the input language. OdinMP/CCp only supports C as input, while Omni supports C and FORTRAN77. The development language of each frontend is also different, C in Omni and Java in OdinMP/CCp.

There are many projects related to OpenMP, for example, research to execute an OpenMP program on top of the Distributed Shared Memory(DSM) environment on a network of workstations[7], and the investigation of a parallel programming model based on the MPI and the OpenMP to utilize the memory hierarchy of an SMP cluster[9]. Several projects, including OpenMP ARB, have stated the intention to develop an OpenMP benchmark program, though Microbenchmarks[6] is the only one available now.

7 Conclusions

This paper presented an overview of the Omni OpenMP compiler and an evaluation of its performance. The Omni consists of a front-end, an Exc Java tool, and a run-time library, and translates an input OpenMP program to a parallelized C program with run-time library calls. We chose Microbenchmarks and a program in Parkbench to use for our evaluation. While Microbenchmarks measures the performance overhead of each OpenMP construct, the Parkbench program evaluates the performance of array calculation loop parallelized by using the OpenMP programming model. The latter gives some criteria to use to parallelize a program using OpenMP directives.

Our evaluation, using benchmark programs, shows Omni achieves comparable performance to a commercial OpenMP compiler, KAI GuideC, on a SUN system with four processors. It also reveals a problem with the Omni run-time library which indicates that the overhead of thread management data is increased according to the number of processors.

On the other hand, the PGI compiler is faster than the Omni on a COMPaS-II system, and it indicates the optimization of the Omni run-time library could improve its performance, though one must consider the fact that the backend of Omni is different

The evaluation also shows that parallelization using the OpenMP directives is effective and the performance scales up for tiny loops if the loop length is long enough, while the COMPaS-II requires very careful optimization to get peak performance.

References

1. http://www.openmp.org/
2. OpenMP Consortium, "OpenMP Fortran Application Program Interface Ver 1.0", Oct, 1997.
3. OpenMP Consortium, "OpenMP C and C++ Application Program Interface Ver 1.0", Oct, 1998.
4. M. Sato, S. Satoh, K. Kusano and Y. Tanaka, "Design of OpenMP Compiler for an SMP Cluster", EWOMP '99, pp.32-39, Lund, Sep., 1999.
5. C. Brunschen and M. Brorsson, "OdinMP/CCp - A portable implementation of OpenMP for C", EWOMP '99, Lund, Sep., 1999.
6. J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP", EWOMP '99, Lund, Sep., 1999.
7. H. Lu, Y. C. Hu and W. Zwaenepoel, "OpenMP on Networks of Workstations", SC'98, Orlando, FL, 1998.
8. http://www.netlib.org/parkbench/html/
9. F. Cappello and O. Richard, "Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model", PACT '99, pp.108-116, Oct., 1999.
10. http://www.kai.com/
11. http://www.pgroup.com/
12. C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr. and M. Zosel,"The High Performance Fortran handbook", The MIT Press, Cambridge, MA, USA, 1994.
13. John M. Mellor-Crummey and Michael L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", ACM Trans. on Comp. Sys., Vol.9, No.1, pp.21-65, 1991.

This article was processed using the LaTeX macro package with LLNCS style