A portable C compiler for OpenMP V.2.0

Vassilios V. Dimakopoulos
Dept. of Computer Science
University of Ioannina, Greece
E-mail: dimako@cs.uoi.gr

Elias Leontiadis
Dept. of Computer Science
University of Ioannina, Greece
E-mail: ilias@cs.uoi.gr

George Tzoumas
Dept. of Informatics and Telecommunications
University of Athens, Greece
E-mail: grad0491@di.uoa.gr

Abstract

This paper presents an overview of OMPi, a portable implementation of the OpenMP API for C, adhering to the recently released version 2.0 of the standard. OMPi is a C-to-C translator which takes C code with OpenMP directives and produces equivalent C code which uses POSIX threads, similarly to other publicly available implementations. However, in contrast to the latter, OMPi is written entirely in C and, more importantly, implements fully version 2.0 of the OpenMP C API. We present major aspects of the implementation, along with performance results.

1. Introduction

Multiprocessor systems are becoming increasingly popular even for the desktop; workstations with two to four processors are commonplace nowadays. The majority of such systems support shared memory by utilizing either symmetric multiprocessor (SMP) or distributed shared memory (DSM) organizations. Shared memory is even supported in clustered multiprocessors and networks of workstations through software DSM (SDSM).

Shared memory programming is one of the most popular parallel programming paradigms. Until recently, however, writing a shared-memory parallel program required the use of vendor-specific constructs which raised a lot of portability issues. This problem was solved by OpenMP [10, 11], a standard API for C, C++ and Fortran which has been endorsed by all major software and hardware vendors. In contrast to other APIs such as the POSIX threads (*pthreads* for short), OpenMP is a higher level API which allows the programmer to parallelize a serial program in a controlled and incremental way.

The OpenMP API consists of a set of compiler directives for expressing parallelism, work sharing, data environment and synchronization. These directives are added to an existing serial program in such a way that they can be safely discarded by compilers that do not understand the API (thus leaving the original serial program unchanged). That is to say, OpenMP extends but does not change the base language (C/Fortran).

This paper presents OMPi, our implementation of an experimental C compiler for OpenMP version 2.0 and is organized as follows: after reviewing some related implementations in Subsection 1.1, we give an overview of our implementation in Section 2. In Section 3 we evaluate OMPi's performance and compare it to other implementations. We conclude the paper in Section 4 with a summary and a discussion of the project's current status.

1.1. Related work

Most of the compilers that understand the OpenMP directives are vendor proprietary or commercial. However, some non-commercial implementations have recently become available, such as the OdinMP/CCp and the Omni compilers.

OdinMP/CCp [3] supports OpenMP for C and is a C-to-C translator that produces C code with pthreads calls. A new version called OdinMP is available as part of the Intone project [2]. The Intone project aims at producing a compilation system along with instrumentation and performance libraries for OpenMP. The current version of OdinMP [8] produces C/C++ code but it does not fully support the OpenMP API (e.g. there are no `threadprivate` variables allowed). The output code includes calls to the Intone run-time library which is based on the Nanos system.

Nanos [1] was a source-to-source Fortran compilation environment mainly for SGI Irix machines, which included a parallelizing compiler, a user-level thread library and visualization tools. The Nanos system supported a subset of version 1.0 of the OpenMP Fortran standard.

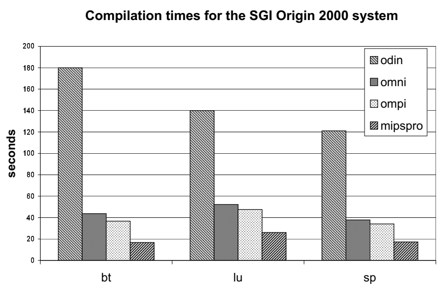
Omni [13] is a sophisticated compilation system that supports OpenMP for both C and Fortran. It is also a source-to-source compiler which can target a number of thread libraries such as pthreads, Solaris threads, IRIX sprocs, etc., and it also includes a cluster-enabled environment. Omni currently adheres to the first version of the C standard.

2. Overview of OMPi

Like the other publicly available implementations, OMPi is a C-to-C translator which takes as input C source code with OpenMP directives and outputs equivalent C code which uses pthreads, ready for parallel execution on a multiprocessor. Pthread-based code is actually a desirable feature since it is quite portable, and can even be used in uniprocessor machines. Our students do most of their program development in uniprocessor workstations and move to multiprocessor machines at the final stages of code finetuning.

In contrast with other implementations, which support the first version of the standard [10], our compiler adheres to the latest (second) version [12]. To the best of our knowledge, this is the first publicly available implementation for this new version of the standard.

In addition, OMPi is implemented entirely in C, while OdinMP and Omni have parts of the compiler written in Java and require a Java interpreter during compilation. We made this choice for many reasons, an important one being compilation performance. First, Java may not be available on all platforms; second its resource requirements are quite high. This shows up clearly during compilation; Fig. 1 includes a comparison of compilation times on a SGI Origin 2000 with 48 CPUs, running Irix 6.5 and on a 2-CPU Compaq Proliant ML570 server running Redhat Linux 9.0. The compilation times are for three applications (bt, lu, sp) from the NAS Parallel Benchmarks suite [7]. We compared OMPi to OdinMP/CCp, Omni and the corresponding proprietary compilers which support OpenMP (MIPSpro 7.3 and Intel C/C++ compiler 7.1). It is seen that while the commercial compilers have an expected advantage, OMPi is faster than the other implementations.



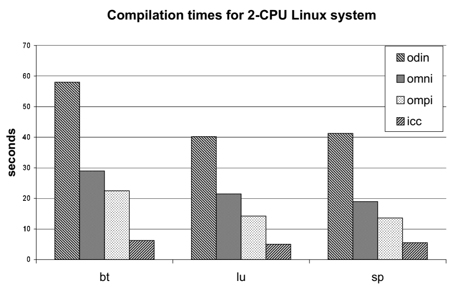


Figure 1. Compilation times for three of the NAS Parallel benchmarks on two systems

2.1. Code parallelization

The parallelization processes consists of various code transformations. In particular, a parallel region spawns a number of threads, each executing the code inside the region. The transformed program has this code moved to a new function which is executed by all threads. Private variables are declared inside this function while shared variables are accessed through pointers.

An example is given in Fig. 2 where:

```
int a;          /* global */

main()
{
  int b, c;

#pragma omp parallel private(c)\
                     num_threads(3)
  {
    c = b  a;
    ...
```

```
 1  int a;

 2  typedef struct       /* Shared vars structure */
 3          {
 4            int (*b);  /* b is shared, non-global */
 5          } par0_t;

 6  main()
 7  {
 8    int b, c;

 9    _omp_initialize();
10    {
11      /* Declares par0_vars, the shared var struct */
12      _OMP_PARALLEL_DECL_VARSTRUCT(par0);
13      /* par0_vars->b will point to real b */
14      _OMP_PARALLEL_INIT_VAR(par0, b);
15      /* Run the threads */
16      _omp_create_team(3, _OMP_THREAD, par0_thread,
17                       (void *) &par0_vars);
18      _omp_destroy_team(_OMP_THREAD->parent);
19    }
20  }

21  void *par0_thread(void *_omp_thread_data)
22  {
23    int _dummy = _omp_assign_key(_omp_thread_data);
24    int (*b) = &_OMP_VARREF(par0, b);
25    int c;

26    c = (*(b)) + a;
27    ...
28  }
```

Figure 2. Code transformation: original (left), produced (right)

1. Each thread will execute `par0_thread()`, lines 16 and 21–28.
2. Private variables (like `c`) are re-declared as local to `par0_thread()`, line 25.
3. Shared variables that are local in `main()` (like `b`) cannot be accessed in `par0_thread()`, so a structure is defined which contains pointers to all shared variables associated with the parallel region (`par0_t par0_vars` in the example, lines 12 and 2–5).
4. Global variables are by nature available to all threads. Thus, shared global variables (like `a`) are not included in the structure, and are accessed directly by each thread (line 25).

Special precautions are taken for array variables, for which we have noticed that some implementations (like OdinMP/CCp 1.02) have problems.

OMPi initially creates a pool of threads which are put to sleep, waiting for work. Upon encountering a parallel region, a number of threads are awakened and are given the function to execute (along with the master thread). At the end of the parallel region each threads goes to sleep; after all threads sleep, the master thread is the only one to continue its execution.

The number of threads to participate in the execution of a parallel region is governed by the standard OpenMP library functions and environmental variables and can also be dynamically adjusted. The new clause `num_threads(N)` of OpenMP V.2.0 is also supported whereby the parallel region is enforced to use exactly N threads.

2.2. Data environment

The handling of shared and private variables was described in the previous section. In addition, re-privatization of variables is supported. That is, variables declared as private in a parallel region can be declared as private in a nested directive, as in the following example:

```
int a;

#pragma omp parallel private(a)
{
  ...
  #pragma omp for private (a)
  ...
}
```

OMPi supports `threadprivate` variables, where globally defined variables can be local to the executing threads. In conformance with V.2.0, static block-scope variables are also allowed to be declared as `threadprivate`.

Finally, the new `copyprivate` clause is implemented for `single` directives, whereby upon completion of the `single` region the value of certain variables are broadcast to all threads. In OMPi the thread that executed the `single` region allocates sufficient space on a designated area of the team's master thread and copies the required data. The rest of the threads copy the data in parallel from the master thread to their private space right after the implied barrier at the end of the `single` region. The last thread to do so frees the allocated space.

2.3. Implementation

OMPi is implemented entirely in C, using the standard `flex` and `bison` tools for lexical and syntax analysis correspondingly. This makes it quite portable and porting it to different systems has been proved a trivial task. It currently runs on a variety of uniprocessor and multiprocessor Sun Solaris, SGI Irix and Linux machines supporting the pthreads library.

OMPi includes the compiler itself and a supporting run-time library. The compilation process of a source file is performed in three steps. In the first step the C preprocessor is invoked. In the second step, the resulting file undergoes code transformations and produces C code with calls to pthreads and the run-time library. In the third step the the host's native C compiler is invoked to produce the executable.

3. OMPi performance

We have invalidated our implementation and evaluated its performance using a multitude of application and benchmark codes. Here we present the results for the NAS Parallel Benchmarks suite (NPB, [7]), version 2.3, which has been ported to OpenMP C by the Omni group [9].

The benchmarks were run on two different systems. The first one was an SGI Origin 2000 machine (Irix 6.5) with a total 48 MIPS R10000 CPUs, where we only had access to 8 of them. The other one was a SUN E-1000 server with 4 Sparc CPUs and Solaris 5.7. In the first system we had access to the native MIPSpro V.7.3 compiler which supports OpenMP pragmas, while the second system did not have a native OpenMP compiler. We also used Omni and tried to use OdinMP/CCp but the later had a lot of problems, while it could not be used in the SUN machine due to the lack of Java. All benchmarks were Class W.

Figs. 3–4 show the results for a sample of 4 out of the 8 application codes, namely the BT, CG, FT and LU routines. The results in Fig. 3 correspond to the SGI machine while the ones in Fig. 4 are for the SUN machine. For the FT benchmarks the Omni compiler failed for 6 or more threads and we were unable to resolve the problem. From the plots it is easily seen that OMPi outperforms Omni in most cases while its performance is comparable with that of the MIPSpro compiler on the SGI system.

3.1. EPCC microbenchmarks

We have used the EPCC microbenchmarks [4] from the University of Edinburgh to measure the synchronization and loop scheduling overheads incurred in by the OpenMP directives.

In Fig. 5 we give the results on the SGI machine and we compare it with OdinMP/CCp. We note that we encountered problems with getting reliable measurements with the MIPSpro compiler which we could not resolve and thus could not include results here. It is seen that OMPi scales reasonably well, and in the same spirit as the Fortran results reported on a similar machine [5], while OdinMP/CCp shows poor performance. The only benchmark that seems to scale poorly in OMPi is the `ordered` one which we are currently investigating.

In Fig. 6 we give the results on the SUN machine, compared to corresponding results by Omni. Omni shows higher overheads which however remain more or less constant. OMPi incurs smaller overhead, but the `parallel` and `reduction` microbenchmarks incur slightly higher overhead for 4 threads.

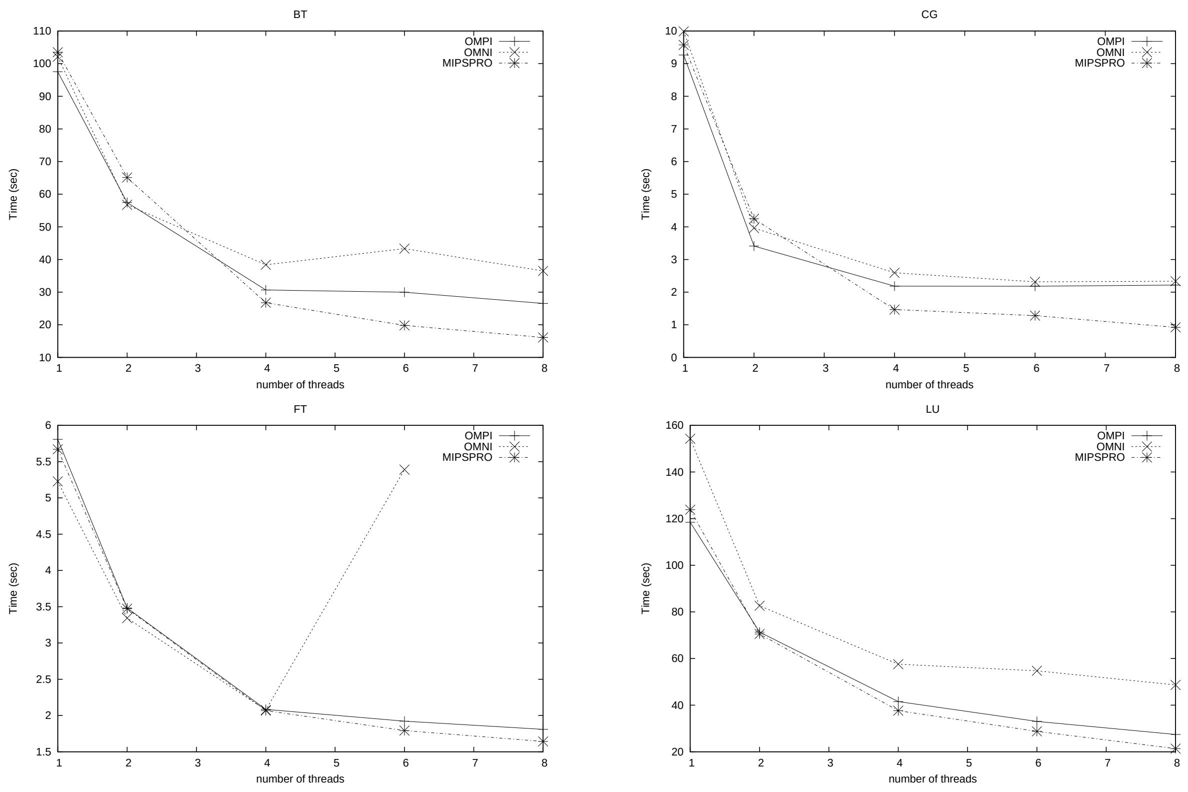


Figure 3. NPB benchmarks on the SGI Origin 2000

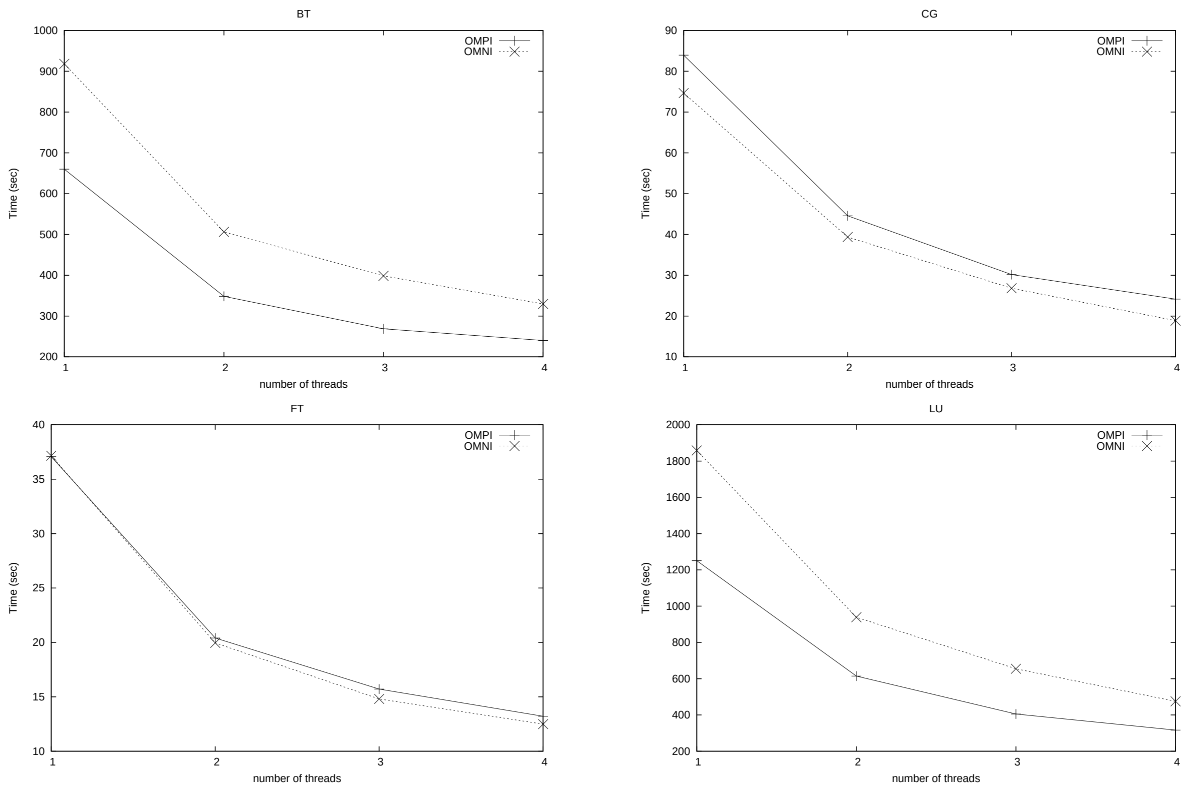


Figure 4. NPB benchmarks on the SUN E-1000

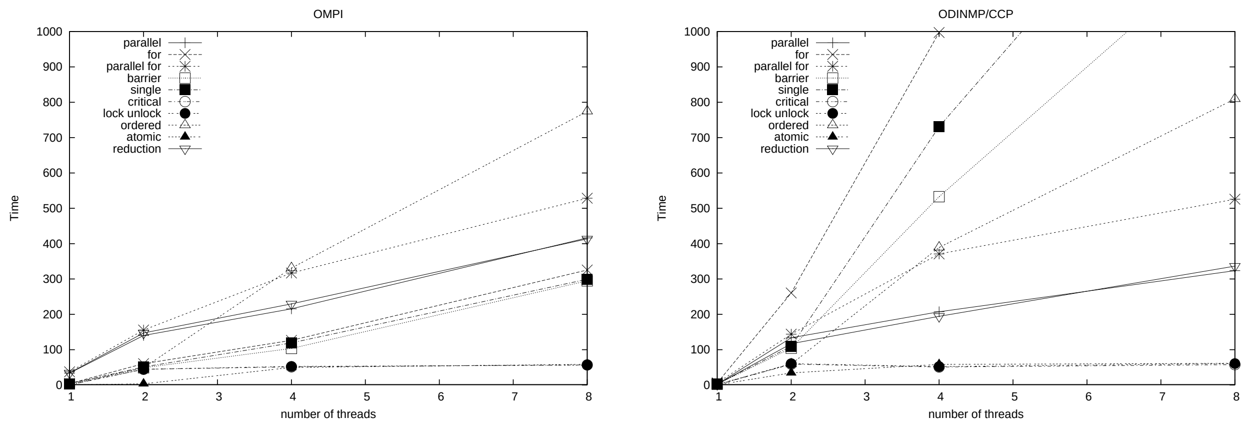


Figure 5. EPCC microbenchmarks on the SGI system

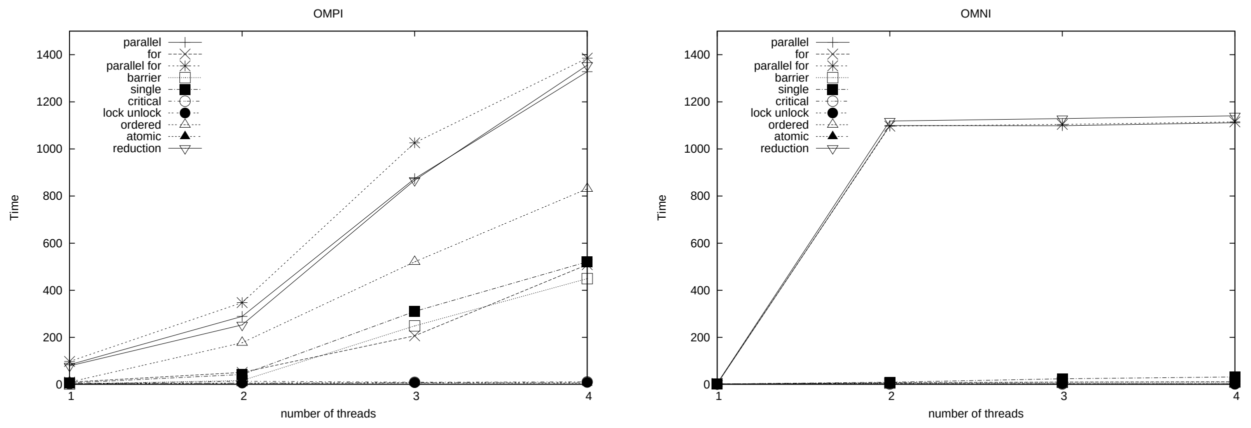


Figure 6. EPCC microbenchmarks on the Sun system

We are currently investigating the microbenchmarks results to determine the sources of overhead so as to be able to improve OMPi-produced code's scalability.

4. Discussion

OMPi is an experimental implementation of the OpenMP API for C, implemented entirely in C, which uses the standard POSIX threads library. Its performance has been proved quite satisfactory, generally being comparable or superior to other publicly available implementations, and reasonable as compared to native compilers we had access to.

OMPi adheres to the second version of the OpenMP C API which was released in March 2002 [12] and includes a number of clarifications for the first version as well as a number of new features. The most important new features which are supported in OMPi are:

- `num_threads` clause in the parallel directive
- reprivatization: a private variable in a parallel region is allowed to be marked as private in a nested directive.
- static variables can also be marked as threadprivate
- run-time routines to measure wall-clock time, `omp_get_wtick()` and `omp_get_wtime()`.
- new `copyprivate` clause in the `single` directive whereby the value of a private variable is copied from one thread to the others.

OMPi is a on-going project in the University of Ioannina and currently undergoes major improvements. We are in the process of targeting, apart form POSIX threads, the host's native thread libraries such as solaris threads for SUN machines and sprocs for SGI Irix machines, which we expect will boost OMPi's performance even further.

Also, we are currently adding profiling features in our compiler, in the spirit of POMP [6], and constructing a graphical tool for performance tuning.

OMPi and its source code will be available at the following URL: `http://www.cs.uoi.gr/~ompi`.

References

[1] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro and J. Oliver, "NanosCompiler: A Research Platform for OpenMP Extensions," in *Proc. EWOMP99, the 1st Europ. Worksh. OpenMP,*, Lund, Sweden, Oct. 1999,

[2] M. Brorsson, "Intone – Tools and Environments for OpenMP on Clusters of SMPs," in *Proc. WOMPAT 2000, Worksh. OpenMP Applic. and Tools*, San Diego, CA, USA, July 2000,

[3] C. Brunschen and M. Brorsson, "OdinMP/CCp – A portable implementation of OpenMP for C," *Concurrency: Practice and Experience*, Vol. 12, pp. 1193–1203, Oct. 2000.

[4] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proc. EWOMP 1999, Europ. Worksh. OpenMP*, Lund, Sweden, Sept. 1999,

[5] J. M. Bull and D. O'Neill, "A Microbenchmark Suite for OpenMP 2.0," in *Proc. EWOMP 2001, Europ. Worksh. OpenMP*, Barcelona, Spain, Sept. 2001,

[6] B. Mohr, A. Mallony, H. - C. Hoppe, F. Schlimbach, G. Haab and S. Shah, "A performance monitoring interface for OpenMP," in *Proc. EWOMP 2002, Europ. Worksh. OpenMP*, Roma, Italy, Sept. 2002,

[7] NASA, "The NAS Parallel Benchmarks," *http://www.nas.nasa.gov/Software/NPB/*,

[8] OdinMP website, *http://odinmp.imit.kth.se/*,

[9] Omni OpenMP Compiler website, *http://phase.hpcc.jp/Omni*,

[10] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," *http://www.openmp.org, Version 1.0*, Oct. 1998.

[11] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," *http://www.openmp.org, Version 2.0*, Nov. 2000.

[12] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," *http://www.openmp.org, Version 2.0*, Mar. 2002.

[13] M. Sato, S. Satoh, K. Kusano and Y. Tanaka, "Design of OpenMP Compiler for an SMP Cluster," in *Proc. EWOMP '99, 1st Europ. Worksh. OpenMP*, Lund, Sweden, Sept. 1999, pp. 32–39.