The OMPi OpenMP/C Compiler

Vassilios V. Dimakopoulos and Alkis Georgopoulos

Department of Computer Science
University of Ioannina, Ioannina, Greece
{dimako,alkisg}@cs.uoi.gr

Abstract. We have designed and implemented OMPi, a portable compiler for OpenMP/C. This paper presents an overview of our compiler and its supporting libraries. OMPi is a C-to-C translator that takes C code with OpenMP directives and produces equivalent multithreaded C code ready for execution on a multiprocessor. Our compiler is the only publicly available one that adheres to version 2.0 of the standard. OpenMP uses POSIX threads for portability, but its architecture allows targeting other thread libraries, as well. It includes platform-specific optimizations and a full POMP library implementation for instrumentation and performance monitoring.

1 Introduction

The shared address space (or simply shared memory) model has been traditionally one of the most popular parallel programming paradigms; intimately tied to shared-memory multiprocessor architectures, it can also be supported in distributed memory organizations (e.g. clusters) through distributed shared memory (DSM) mechanisms.

A long list of non-portable, incompatible, vendor-specific environments for shared memory programming has recently been surpassed by OpenMP [14, 15], an application programming interface (API) for C, C++ and Fortran. OpenMP has been endorsed by all major software and hardware vendors and will soon become the de facto standard for shared-memory programming. In contrast to other APIs such as the POSIX threads [5] (*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 and runtime supporting calls. It provides for expressing parallelism, sharing work, specifying the data environment and synchronization. The 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). As a consequence, OpenMP extends but does not change the base language (C/Fortran).

Many commercial compilers nowadays provide support for OpenMP, including compilers by Fujitsu, Intel, PGI, SGI and Sun. Recently, a number of those compilers added support for version 2.0 of OpenMP [16, 15], which defines a number of additions and enhancements to the original version of the API. A number of research compilers have also been reported, all of them supporting OpenMP v.1.0.

OdinMP/CCp [3] supports OpenMP for C and is a standalone C-to-C translator that produces C code with pthreads calls. Another version (called OdinMP) is available as part of the Intone project [2], which aims at producing a compilation system along with instrumentation and performance libraries for OpenMP. OdinMP is a C/C++ translator but it does not fully support the OpenMP API (for example `threadprivate` variables are not allowed).

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 [17] is a sophisticated compilation system that supports OpenMP for both C/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 support for clusters through a software DSM library.

OMPi is our experimental OpenMP/C compiler, developed at the Department of Computer Science of the University of Ioannina. It is an efficient and portable open-source implementation, and the only publicly available one that adheres fully to version 2.0 of the standard for C. OMPi is an on-going project that aims at providing a tool for OpenMP programming and runtime support research. This paper presents an overview of the implementation of OMPi and its support libraries, and is organized as follows: Section 2 provides details about our implementation, including its thread and execution model, its code transformations, and its runtime support. In section 3 we discuss a few non-portable implementation issues of the compiler, indented to improve performance on specific platforms. Section 4 presents the monitoring interface implemented in OMPi. In Section 5 we include sample performance results for the NAS parallel benchmarks, used to evaluate OMPi and compare it to other implementations. We conclude the paper in section 6 with a summary and a discussion of the project's current status.

2 The compiler

OMPi is a source-to-source translator that takes as input C source code with OpenMP V.2.0 directives and outputs equivalent multithreaded C code, ready to be built and executed on a multiprocessor. An initial prototype was presented in [6]. The current version is fully V.2.0 compliant, can target different thread libraries through a unified thread abstraction, and includes many architecture-dependent as well as higher-level optimizations. Finally, the compiler and the runtime libraries include support for the POMP performance monitoring interface.

OMPi is implemented entirely in C, while the majority of the other research compilers have parts written in Java and require a Java interpreter during compilation. It has been ported effortlessly on many different platforms, including Intel / Linux, Sun / Solaris and SGI / Irix systems.

2.1 OMPi threads

OMPi produces multithreaded C code. Its architecture is such that any specific thread library can be supported through a well-defined generic interface, making OMPi quite extensible. This interface consists of an opaque data type for thread objects (`othread_t`) and a set of functions which include calls for creating threads (`othread_create()`), utilizing per-thread private data and manipulating various lock types (e.g. `othread_set_lock()`). Details of the thread interface can be found in [8].

OMPi currently supports two specific thread libraries, both through the generic thread interface. The first is POSIX threads (pthreads). This is OMPi's default thread library target mainly for portability reasons, since it is available almost everywhere. Moreover, recent pthreads implementations have been highly optimized in various environments (e.g. for Linux kernels 2.6.x [7]), which make pthreads attractive performance-wise, too. The second library can be used in Sun / Solaris machines, where the user has the option of producing code with Solaris threads [19] calls.

It should be noted that the generic interface provides for a transparent choice of the underlying thread library. That is, the source code is not affected in any way — the actual thread library that will be used is only included at linking time.

2.2 Execution model

OpenMP uses the fork/join model. When a `parallel` region is encountered in the code, thread entities are created to execute the specified task, and are destroyed (joined) at the end of the region.

For performance reasons, OMPi does not create / destroy threads at each `parallel` region but recycles them instead. Upon initialization, OMPi creates a pool of threads equal to the number of available processors in the system, which are put immediately to sleep, waiting for work. Whenever a `parallel` region is encountered, a number of threads are awakened and are given a function to execute, with the master thread participating, too. At the end of the region each thread goes back 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.

2.3 Source transformations

The compilation process involves extensive transformations to the source code which produce the final multithreaded C file. In particular, whenever a `parallel` region is encountered, the following transformation process is followed:

- The code inside the `parallel` region is moved to a new function which will be called by all threads.
- In the place of the original code, code for the creation of a team of threads is injected. The threads are drawn from of the thread pool, and each thread executes the new function. Right after, all threads are joined and returned to the pool.
- Private variables are re-declared as local inside the new function.
- Shared variables that are global in scope need no special treatment since they are by nature available to all threads.
- Non-global shared variables are declared locally as pointers, initialised to point the original variables.
- Non-global shared variable references are replaced by appropriate pointer references.

Worksharing constructs do not produce any new functions in the final code but require restructuring of the original code to accomodate run-time decisions. For example, a `for` directive is implemented by the following steps:

- A new lectical scope is created.
- The loop index and any reduction variables are declared as local within this scope, along with other utility variables.
- The thread executes a loop asking for the next set of iterations.
- The run-time library, calculates the initial, final and increment values for the next set of iterations and returns them to the calling thread. The calculated values are based on the selected schedule, chunk size and the thread's id.
- The thread executes the `for` loop code with the pre-calculated values and loops back to ask for the next set of iterations.
- The thread that receives the last set of iterations updates the `lastprivate` variables, if any.
- If a `reduction` clause is specified, each thread contributes its local value to the shared reduction variable using locks for mutual exclusion.
- If a `nowait` clause is not present, a barrier call is generated (which may be removed by the compiler if there is another barrier call right after that, for optimization reasons).

2.4 Runtime support

It is beyond the scope of this paper to go into further details, but transformations analogous to the ones in the previous section occur for every OpenMP directive. All transformations and all additionally generated code produce calls to the `libompi` library which provides runtime support. This includes primitives for thread management, for iteration scheduling, for mutual exclusion and for synchronization.

In particular, the pool of threads is managed at runtime by `libompi`, which provides functions (trannsparently used by the compiler) for giving work to threads, suspending them to a waiting queue and resuming them in order to form active teams. The runtime library also includes calls for dynamically distributing iterations among threads, for all types of OpenMP `for` schedules. In addition, the runtime library implements all runtime calls specified by the OpenMP API, such as calls for querying the id of a thread, the number of processors in the system, the wall clock time, etc.

Most parallel programs utilize synchronization primitives, such as barrier calls, which account for a significant percentage of the total execution time. The compiler attempts to speed up the generated code by removing redundant barriers. This is an optimization whose significance is well known [9]. OMPi's runtime library includes a base barrier implementation which is, however, overrideable by the user. The barrier functions have been implemented as `weak symbols`; this allows any user to provide their own custom implementations for the barrier functions without requiring access to the internals of the compiler or its support libraries.

3 Platform-specific issues

OMPi can be easily ported on many different platforms, in fact on any platform which supports pthreads, such as all modern Unix derivatives. The source code of the compiler, as well as the output code it generates is highly portable. However, portability sometimes is a tradeoff for performance. This is why in various platforms we had access to we tried to provide hardware-specific and operating-system specific optimizations.

First, most data structures have been made cache-aligned. For the systems we considered cache lines were 32–64 bytes long. Core structures were also padded up to cache line size so as to avoid false-sharing phenomena. Those two simple platform-dependent considerations alone were enough to boost performance up to 20% in various benchmarks and more than 5% in some applications.

Another characteristic platform-specific issue is the implementation of the `flush` directive which has not been discussed widely in the open literature. The `flush` primitive is implied in almost all OpenMP directives and basically provides a mechanism for enforcing sequential memory consistency behavior on machines that support relaxed memory ordering models. However, the `flush` primitive can be a quite demanding operation with significant performance impact.

A portable but inefficient implementation involves locking and unlocking a pthreads mutex, which provides for the desired behaviour [5]. We also resort to this method, but for the machines we had access to we utilize faster methods. In particular, we make use of CPU-specific machine instructions that force the correct memory orderings. For IA32 and Sparc CPUs, the utilized instructions are summarized in Table 1.

Table 1. Memory ordering instructions used by OMPi for some CPUs

Instruction	*CPU*	*Ordering Model*
`STBAR`	Sparc V8	PSO
`MEMBAR`	Sparc V9	TSO/PSO/RMO
`XCHG`	Intel Pentium	PC
`CPUID`	Intel Pentium	PC

4 Performance monitoring

There has been a long standing demand for an official performance monitoring specification for OpenMP. Although there has been no official endorsement yet, there exists a first incomplete proposal [11] which tries to combine two independent proposals, POMP [12] and OMPI, which is part of the Intone project. The two interfaces are fundamentally different and it is questionable whether a unified interface can emerge. Of the two, POMP seems the faster, better-designed and closer to the proposal in [11].

OMPi provides full support for the POMP interface. Upon user request (by passing specific parameters to the compiler), the produced code goes through additional transformations which automatically insert POMP calls at specific places, including entry and exit points of:

- parallel blocks
- for, sections, single and master blocks
- explicit and implicit barriers
- critical regions

All runtime library OpenMP functions include monitoring calls and there are also provisions for user-driven monitoring of arbitrary regions of the program. Finally, because of the overhead induced by the performance monitoring mechanisms, and the size of trace data collected, POMP includes directives that can turn profiling on / off anywhere in the user program.

POMP, used in conjunction with visualization utilities constitutes a powerful tool for parallel program analysis. We have used the EPILOG library [12] to intercept the POMP calls that OMPi produces and targeted the metrics to the CUBE visualization utility [18].

In Fig. 1 we provide a sample performance output which was obtained after compiling with OMPi and executing a simple program which calculates $\pi = 3.14\cdots$. CUBE was used to display the collected traces. The following is a portion of the program:

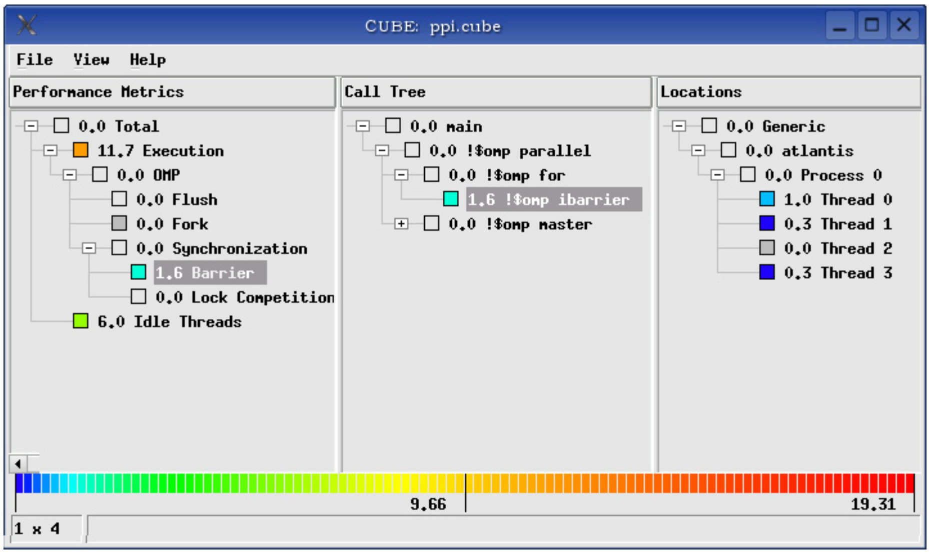

Fig. 1. CUBE display of collected POMP performance data

```
#pragma omp parallel
{
  #pragma omp for reduction(+: pi)
    for (i = 0; i < N; i++)
      pi += 4.0 / (1.0 + (i+0.5)*(i+0.5)*w*w;
  #pragma omp master
  {
    printf("pi = \%1.20lf", pi*w);
    #pragma omp inst begin(sleeping)
    sleep(2);
    #pragma omp inst end(sleeping)
  }
}
```

The left column displays performance metrics for various entities and the middle column displays the call tree for a user-selected metric of the left column. In this example, the total execution time was 11.7 sec of pure computation, plus 1.6 sec spent on a barrier synchronization plus another 6 sec spent because of idling (in the `sleeping` region of the above code). The barrier on which threads spent their time, is the implicit one at the end of the `master` region. In the third column, per-thread performance data are displayed. In particular, the barrier in question made thread 0 (the master thread) spent a whole second waiting, while the others waited much less. This is a clear indication of load imbalance. A `guided` for schedule could possibly lead to better performance.

5 OMPi performance

OMPi has been validated through countless tests, including the OpenMP validation suite reported in [10], which tests for implementation correctness and compliance to the API specifications. Numerous benchmarks, microbenchmarks [4] and application codes have also been used to assess the performance of our compiler. Here, and due to space limitations, we present indicative results for the NAS Parallel Benchmarks suite (NPB, [13]), version 2.3, which has been ported to OpenMP C by the Omni group.

The benchmarks were compiled and executed on two different systems: an SGI Origin 2000 machine running Irix 6.5 with a total of 64 MIPS R10000/12000 CPUs (where we only had access to 8 of them) and a Compaq Proliant ML570 server with 4 Intel III Xeon CPUs running Linux. In the SGI machine we had access to the native MIPSpro V.7.3 compiler, while the Compaq machine offered the ICC Intel compiler, both supporting OpenMP pragmas.

Except OMPi and the commercial compilers, we used Omni and tried to use OdinMP/CCp but the later was quite unstable. All benchmarks were class W.

Fig. 2 shows the results for a sample of 4 out of the 8 application codes, namely the BT, CG, FT and LU routines in the SGI machine. For the FT benchmarks the Omni compiler failed for 6 or more threads and we were unable to resolve the problem. Fig. 3 show the results for the same set of applications in the Compaq machine. We were not able to execute the FT benchmark reliably with the Intel compiler for more than one thread, and this is why it is not included in the corresponding plot in Fig. 3.

From the plots it is easily seen that OMPi outperforms Omni in most settings. In addition, itsreleasing performance is in many cases comparable with that of the commercial compilers. One important conclusion drawn from our experimentations is that most research compilers suffer from problems related to stability and conformance to the standard.

6 Conclusion

OMPi is an experimental OpenMP source-to-source compiler for C, that adheres to the latest V.2.0 of the API. It has been developed in the University of Ioannina and has reached a maturity and stability level that allows us to utilize it as an educational tool in our Parallel Processing course both at the undergraduate and the graduate level.

Performance-wise, it generates quite satisfactory code, which is in general comparable or superior to other publicly available implementations, and reasonable as compared to the native compilers we had access to.

Its most important strength, though, is its open and extensible architecture which makes it an ideal research tool on parallel programming, runtime support and synchronization mechanisms.

We are actively improving and extending OMPi's features. Our next major goal is the support of nested parallelism, whereby `parallel` regions inside other `parallel` regions are allowed to spawn new threads. We are also

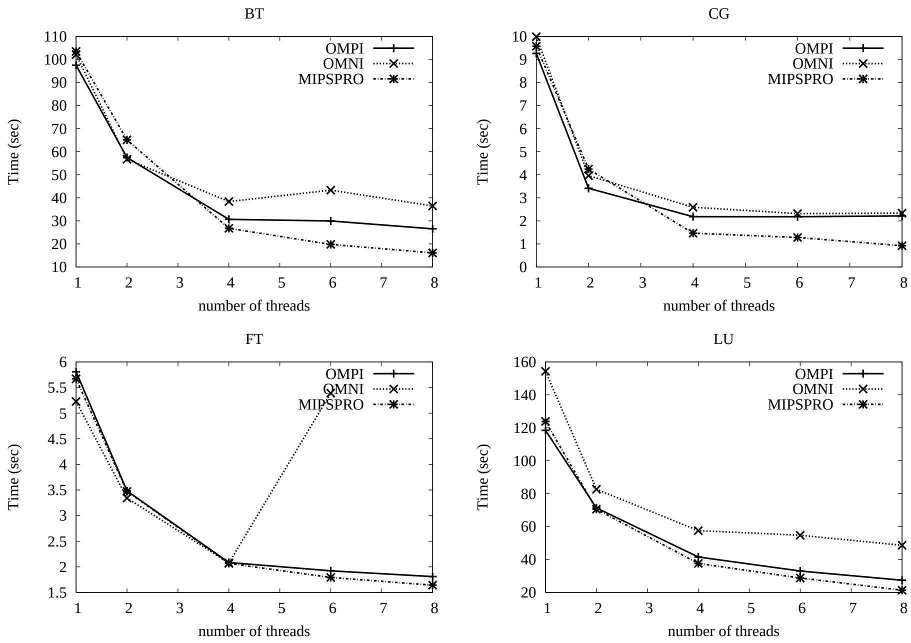

Fig. 2. NPB benchmarks on the SGI Origin 2000

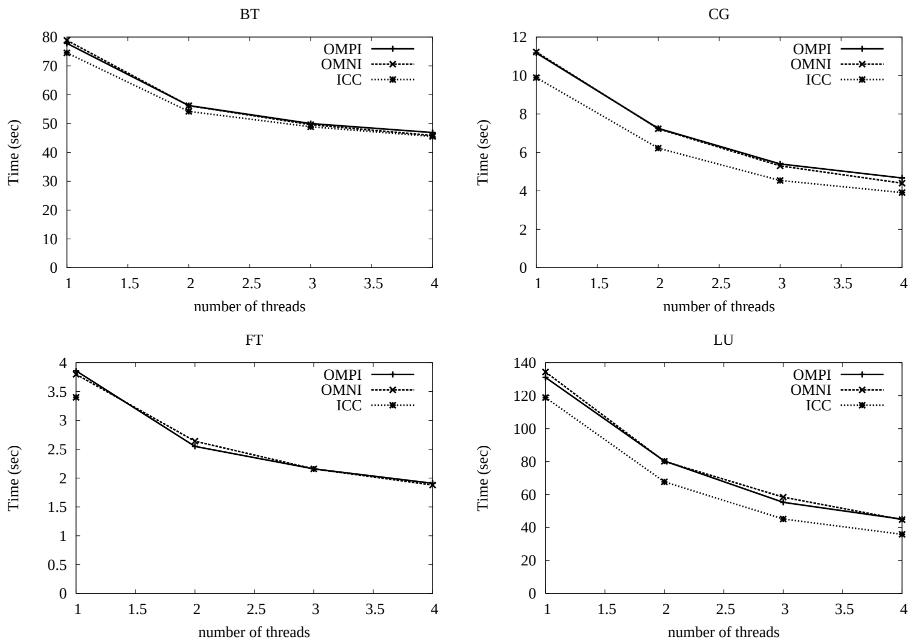

Fig. 3. NPB benchmarks on the Intel Xeon server

working on targeting DSM libraries so as to allow pure OpenMP programming over clusters. OMPi and its source code is available at the following URL: `http://www.cs.uoi.gr/~ompi`.

References

1. Ayguade, E., Gonzalez, M., Labarta, J., Martorell, X., Navarro, N., Oliver, J.: NanosCompiler: A Research Platform for OpenMP Extensions, In: EWOMP99, the 1st Europ. Worksh. OpenMP, Lund, Sweden (1999)
2. Brorsson, M.: Intone – Tools and Environments for OpenMP on Clusters of SMPs, In: WOMPAT 2000, Worksh. OpenMP Applic. and Tools, San Diego, CA, USA (2000)
3. Brunschen, C., Brorsson, M.: OdinMP/CCp – A portable implementation of OpenMP for C. Concurrency: Practice and Experience **12** (Oct. 2000) 1193–1203
4. Bull, J. M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0, In: EWOMP 2001, Europ. Worksh. OpenMP, Barcelona, Spain (2001)
5. Butenhof, D. R.: Programming with POSIX Threads. Addison-Wesley 1997
6. Dimakopoulos, V. V., Leontiadis, E., Tzoumas, G.: A Portable C Compiler for OpenMP V.2.0, In: EWOMP 2003, 5th European Workshop on OpenMP, Aachen, Germany (2003)
7. Drepper, U., Molnar, I.: The Native POSIX Tread Library for Linux. RedHat White Paper. (Feb. 2003)
8. Georgopoulos, A.: Implementation Issues for an OpenMP Compiler (in Greek). MSc Thesis, Dept. of Computer Science, University of Ioannina. (Sept. 2004)
9. Müller, M.: Some Simple OpenMP Optimization Techniques, In: WOMPAT 2001, Worksh. OpenMP Applic. and Tools, West Lafayette, Indiana (2001)
10. Müller, M., Neytchev, P.: An OpenMP Validation Suite, In: EWOMP 2003, 5th Europ. Worksh. OpenMP, Aachen, Germany (2003)
11. Mohr, B., Mallony, A., Hoppe, H. - C., Schlimbach, F., Haab, G., Shah, S.: A performance monitoring interface for OpenMP, In: EWOMP 2002, Europ. Worksh. OpenMP, Roma, Italy (2002)
12. Mohr, B., Mallony, A., Shende, S., Wolf, F.: Design and Prototype of a Performance Tool Interface for OpenMP. The Journal of Supercomputing **23** (2002) 105–128
13. NASA: The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/
14. OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface. http://www.openmp.org, Version 1.0 (Oct. 1998)
15. OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface. http://www.openmp.org, Version 2.0 (Nov. 2000)
16. OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface. http://www.openmp.org, Version 2.0 (Mar. 2002)
17. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP Compiler for an SMP Cluster, In: EWOMP '99, 1st Europ. Worksh. OpenMP, Lund, Sweden (1999) 32–39
18. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An Algebra for Cross-Experiment Performance Analysis, In: ICPP'04, Int'l Conference on Parallel Processing, Montreal, Quebec, Canada (2004) 63–72
19. Sun Microsystems Inc: Solaris Multithreaded Programming Guide. Prentice Hall 1996