Portable Compilers for OpenMP*

Seung Jai Min, Seon Wook Kim**, Michael Voss, Sang Ik Lee, and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
http://www.ece.purdue.edu/ParaMount

Abstract. The recent parallel language standard for shared memory multiprocessor (SMP) machines, OpenMP, promises a simple and portable interface for programmers who wish to exploit parallelism explicitly. In this paper, we present our effort to develop portable compilers for the OpenMP parallel directive language. Our compiler consists of two parts. Part one is an OpenMP parallelizer, which transforms sequential languages into OpenMP. Part two transforms programs written in OpenMP into thread-based form and links with our runtime library. Both compilers are built on the Polaris compiler infrastructure. We present performance measurements showing that our compiler yields results comparable to those of commercial OpenMP compilers. Our infrastructure is freely available with the intent to enable research projects on OpenMP-related language development and compiler techniques.

1 Introduction

Computer systems that offer multiple processors for executing a single application are becoming common place from servers to desktops. While programming interfaces for such machines are still evolving, significant progress has been made with the recent definition of the OpenMP API, which has established itself as a new parallel language standard. In this paper, we present a research compiler infrastructure for experimenting with OpenMP.

Typically, one of two methods is used for developing a shared-memory parallel program: Users may take advantage of a restructuring compiler to parallelize existing uniprocessor programs, or they may use an explicitly parallel language to express the parallelism in the application. OpenMP is relevant for both scenarios. First, OpenMP can be the target language of a parallelizing compiler that transforms standard sequential languages into parallel form. In this way, a parallelizer can transform programs that port to all platforms supporting OpenMP. A first contribution of this paper is to describe such a portable parallelizer. Second, programs written in OpenMP can be compiled for a variety of computer systems, providing portability across these platforms. Our second contribution

* This work was supported in part by NSF grants #9703180-CCR and #9872516-EIA.
** The author is now with Kuck & Associates Software, A Division of Intel Americas, Inc., Champaign, IL 61820.

is to present such an OpenMP compiler, called PCOMP (Portable Compilers for OpenMP), which is publicly available.

Both compilers are built on the Polaris compiler infrastructure [1], a Fortran program analysis and manipulation system. The public availability of PCOMP and its runtime libraries makes it possible for the research community to conduct such experiments as the study of new OpenMP language constructs, detailed performance analysis of OpenMP runtime libraries, and the study of internal compiler organizations for OpenMP programs. In addition, this paper makes the following specific contributions:

- We describe an OpenMP *postpass* to the Polaris parallelizing compiler, which generates parallel code in OpenMP form.
- We describe a compiler that translates OpenMP parallel programs into thread-based form. The thread-based code can be compiled by a conventional, sequential compiler and linked with our runtime libraries.
- We present performance measurements showing that PCOMP yields results comparable to the commercial OpenMP parallelizers.

We know of two related efforts to provide a portable OpenMP compiler to the research community. The Omni OpenMP compiler [2] translates C and Fortran programs with OpenMP pragmas into C code suitable for compiling with a native compiler linked with the Omni OpenMP runtime library. Also the compiler provides a cluster-enabled OpenMP implementation on a page-based software distributed shared memory. The OdinMP/CCp (C to C with pthreads) [3] is also a portable compiler for C with OpenMP to C with POSIX thread libraries. Both efforts are related to our second contribution.

In Section 2 we present an overview of our OpenMP compiler system. Section 3 and Section 4 describe the details of our OpenMP parallelizer, which generates parallel code with OpenMP directives, and the OpenMP compiler, which translates OpenMP parallel code into thread-based form. In Section 5 we measure the performance of PCOMP and other OpenMP compilers. Section 6 concludes the paper.

2 Portable OpenMP Compilers

Figure 1 gives an overview of our OpenMP compiler system. A program can take two different paths through our OpenMP compiler system: (1) A serial program is analyzed for parallelism by the Polaris Analysis passes and then annotated with OpenMP directives by the OpenMP postpass. The output is an OpenMP parallel program. (2) OpenMP parallel programs can be processed by the OpenMP directive parser and then fed to the MOERAE postpass [4], which transforms them into thread-based code. In this scenario, no additional parallelism is recognized by the Polaris Analysis passes. The generated code can be compiled by a sequential compiler and linked with our MOERAE microtask library.

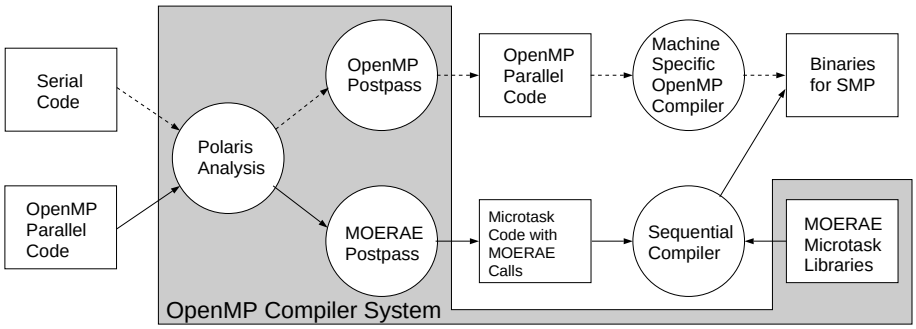

Fig. 1. Overview of our OpenMP compiler system. The same infrastructure can be used to (1) translate sequential programs to OpenMP parallel form and (2) generate code from OpenMP parallel programs. In scenario two, the code is translated into a microtask form, making calls to our MOERAE runtime library.

3 Generating OpenMP from Sequential Program

The Polaris compiler infrastructure is able to detect loop parallelism in sequential Fortran programs. To express this parallelism in OpenMP, we have added an *OpenMP postpass*. It can read the Polaris internal program representation and emit Fortran code annotated with OpenMP directives.

The implementation of this postpass is a relatively straightforward mapping of the internal variables in Polaris to their corresponding directives [5]. Only in the case of array reductions and in privatizing dynamically allocated arrays, more involved language-specific restructuring is done. All other major code restructuring is already implemented in Polaris in a language-independent fashion.

Minor issues arose in that the parallel model used by OpenMP requires some changes to the internal structure of Polaris. Polaris, being tightly coupled to the loop-level parallelism model, lacks a clear method for dealing with parallel regions and parallel sections. Polaris-internally, program analysis information is typically attached to the statement it describes. For example, a loop that is found to be parallel has the DO statement annotated with a *parallel assertion*. The statement is likewise annotated with the variables that are to be classified as shared, private and reduction within the loop nest. In all loop-oriented parallel directive languages that Polaris can generate, this is an acceptable structure.

In order to represent OpenMP parallel regions we have added loop preambles and postambles. Preambles and postambles are code sections at the beginning and end of a loop, respectively, that are executed once by each participating thread. It is now the statement labeled at the beginning of the preamble, which must contain the annotations for parallelism. This statement may no longer be the DO statement. The variables classified as reduction variables, still have to be associated with the DO loop itself. The specification does not require that reductions occur inside loops. You can reduce across parallel regions. These

changes in the structure of where information is to be stored, required a review of the Polaris passes.

4 Compiling OpenMP Parallel Programs with PCOMP

PCOMP (Portable Compiler for OpenMP) translates OpenMP parallel programs into thread-based programs. Two capabilities had to be provided to implement PCOMP. One is an extension of the parser for OpenMP directives, and the other is a translator that converts the Polaris-internal representation into the parallel execution model of the underlying machine. As a target we use a thread-based, microtasking model, as is common for loop parallel execution schemes. The Polaris translation pass into thread-based forms has already been implemented in the form of the MOERAE system, described in [4]. MOERAE includes a runtime library that implements the microtasking scheme on top of the portable POSIX-thread libraries.

We added parser capabilities to recognize the OpenMP directives and represent their semantics in the Polaris internal program form. The OpenMP directives are translated in the following manner:

4.1 Parallel Region Construct

The `PARALLEL` and `END PARALLEL` directives enclose a parallel region, and define its scope. Code contained within this region will be executed in parallel on all of the participating processors.

Figure 2 illustrates parallel region construct translation. PCOMP *always* replaces these directives with a parallel `DO` loop whose lower-bound is a constant 1 and upper-bound is the number of participating threads. Figures 2 (a) and (b) show the code transformation from the source to PCOMP intermediate form for a parallel region construct. A variable *mycpuid* indicates the thread identifier number and *cpuvar* denotes the number of participating threads. The newly inserted loop is asserted as `PARALLEL`, and it allows the MOERAE system to generate the thread-based code shown in Figure 2 (c). At the beginning of the program, the function *initialize_thread* is inserted to initialize the execution environment, such as setting the available number of threads and creating threads. After this initialization, the created threads are in spin-waiting status until function `scheduling` is called.

The function `scheduling` invokes the runtime library to manage the threads and copies shared arguments (`parameters` in Figure 2) into the child threads. Our runtime library provides different variants of this function for the different scheduling options. Currently, only static scheduling is supported.

There are several directive clauses, which may be included on the same line as the `PARALLEL` directive. They control attributes of the region. The directive clauses are translated as follows.

The `IF(`*expression*`)` clause will cause the parallel region to be executed on a single thread, if the expression evaluates to FALSE. This directive will cause a two-version loop to be generated; one loop is serial and the other is parallel.

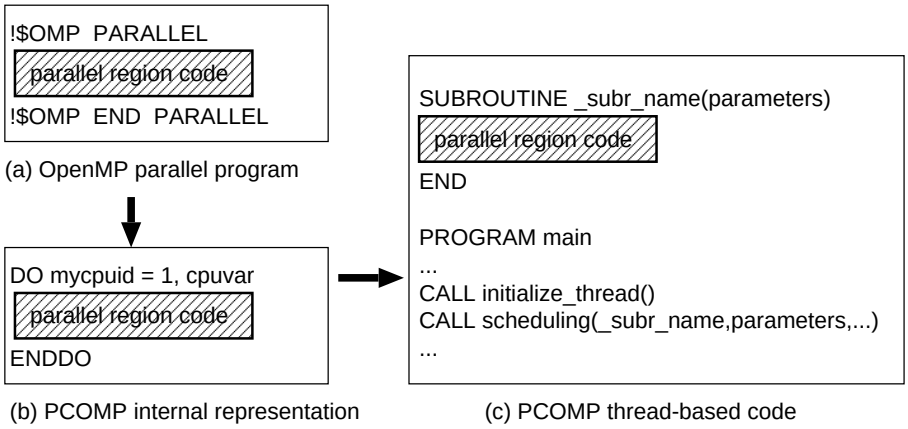

Fig. 2. OpenMP PARALLEL construct transformation. The parallel region is surrounded by a loop whose lower-bound is a constant 1 and upper-bound is the number of participating threads.

The `SHARED(`*variable_list*`)` clause includes variables that are to be shared among the participating threads.

The `PRIVATE(`*variable_list*`)` clause includes variables that are local to each thread, and for which local instances must exist on each participating thread.

The `LASTPRIVATE(`*variable_list*`)` clause is similar to `PRIVATE` directive. The difference is that when the `LASTPRIVATE` clause appears on a `DO` directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. PCOMP uses two-version statements to conditionally perform the updates to the variables if the iteration is the last one.

The `REDUCTION(`*variable_list*`)` directive clause contains those scalar variables that are involved in a scalar reduction operation within the parallel loop. The Polaris program representation already contains fields with this semantics. PCOMP transforms the OpenMP clauses into these fields. The parallel reduction transformation is then performed using the techniques described in [6]. Preamble and postamble code is generated to initialize a new, private reduction variable and to update the global reduction variable, respectively. The update is performed in a critical section using a lock/unlock pair.

4.2 Work-Sharing Construct

The `DO` and `END DO` directives enclose a DO loop inside a region. The iteration space of the enclosed loop is divided among the threads. As an example, the translation process of the `DO` directive with a `REDUCTION` clause is described in Figure 3. The PCOMP OpenMP parser reads `Work-Sharing` directives and assert them to the corresponding `DO` statement. In Figure 3 (b), there are two `DO` statements. The first one is a parallel `DO` loop, which is created by the translation of the `PARALLEL` construct. The directive clauses, such as `PRIVATE` and

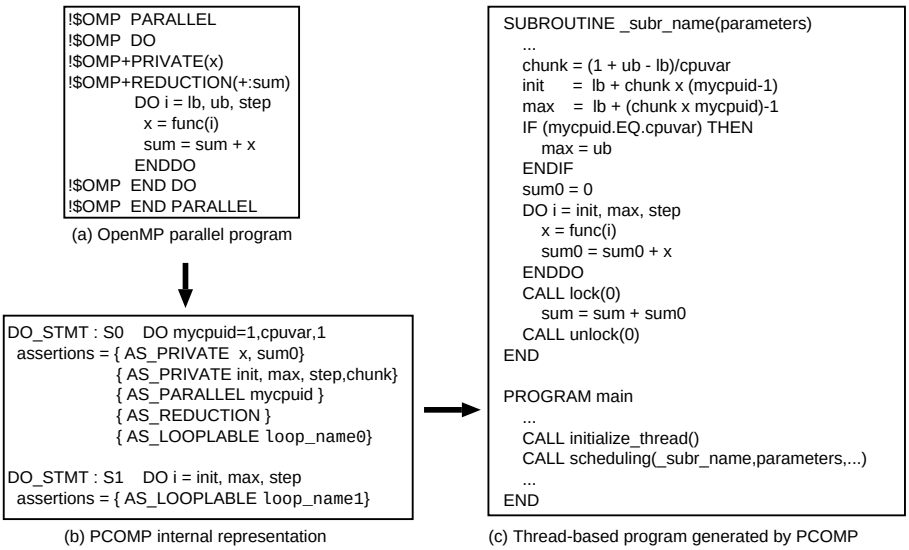

Fig. 3. OpenMP PARALLEL REDUCTION DO construct transformation

`REDUCTION` are transformed to the assertion type information. These assertions give the PCOMP postpass information to generate thread-based code. The second `DO` statement is the original `DO` statement whose loop bound indices will be modified to share iteration space by the participating threads.

Figure 3(c) depicts how the iteration space is divided using loop bound modification. The loop bounds are adjusted to reflect the chunk of iterations assigned to each thread. The *if* statement that compares *mycpuid*, a thread number with *cpuvar*, the number of threads detects the last iteration and adjust the upper bound of the loop to come up with the case when the total number of iterations is not a multiple of the number of threads. PCOMP generates postamble code for `REDUCTION` clauses and encloses the postamble code with `lock` and `unlock` functions implemented in the MOERAE runtime libraries [4].

Synchronization Constructs Critical blocks are surrounded by `CRITICAL` and `END CRITICAL` directives. The code enclosed in a `CRITICAL/END CRITICAL` directive pair will be executed by only one thread at a time. We replace the directives with `lock` and `unlock` functions. The `BARRIER` directive synchronizes all the threads in a team. When encountered, each thread waits until all of the other threads in that team have reached this point. PCOMP replaces the `BARRIER` directive with a `sync` function, which is a runtime library function.

The current implementation of PCOMP supports a subset of the OpenMP 1.0 specification for Fortran. Among the unsupported constructs are parallel sections, flushing operation in critical section and dynamic scheduling for parallel loops.

5 Performance Results

We used benchmark programs (`WUPWISE`, `SWIM`, `MGRID`, and `APSI`) from the SPEC CPU2000 [7] and the SPEComp2001 benchmark suite to evaluate the performance of our compiler. We generated executable code using the options `f95 -fast -stackvar -mt -nodepend -xvector=no -xtarget=ultra2 -xcache=16/32/1:4096/64/1`. The thread-based codes by PCOMP are linked with the MOERAE runtime libraries. For comparison, we also compiled each OpenMP code with the SUN Forte 6.1 OpenMP compiler. We ran the codes on a SUN Enterprise 4000 (Solaris 2.6) system [8] using *ref* data sets.

First, we parallelized the SPEC2000 applications using the Polaris parallelizing compiler [1], which generated the OpenMP codes using our OpenMP postpass. We generated two executable codes: (1) using the SUN parallel compiler, and (2) using a sequential compiler to compile the translated code by PCOMP and link with the runtime library. Figure 4 shows the speedup of these codes relative to the serial execution time of the original codes. The figure shows that our compiler performs similarly to the commercial compiler. The super-linear speedup in `SWIM` is due to a loop interchange in the `SHALOW_DO3500` loop by the parallelizer. In `MGRID` the performance of our compiler is better than Forte 6.1 in all loops except in `RESID_DO600`, where our thread-based postpass generates inefficient code handling `LASTPRIVATE` variable attributes. Figure 4 shows that the Polaris-generated codes successfully exploit parallelism in two of the four benchmarks.

We used an early version of the SPEComp2001 benchmarks for measuring the performance of our PCOMP compiler. Figure 5 shows the speedup of these codes relative to the one-processor execution time of the code generated by the SUN compiler. The figure shows that our compiler performs similarly to the commercial compiler.

6 Conclusion

We presented an OpenMP compiler system for SMP machines. The compilers are built using the Polaris infrastructure. The system includes two compilers for translating sequential programs into OpenMP and for compiling OpenMP programs in a portable manner (PCOMP), respectively.

We showed that the performance of PCOMP is comparable to commercial OpenMP compilers. Our infrastructure is publicly available, enabling experimental research on OpenMP-related language and compiler issues. The Polaris infrastructure has already been widely used in parallelizing compiler research projects. The availability of an OpenMP compiler component now also supports this important, emerging standard in parallel programming languages. Currently, only Fortran77 is supported. Extensions for Fortran90 and C are being developed, providing a complete open-source compiler environment for OpenMP.

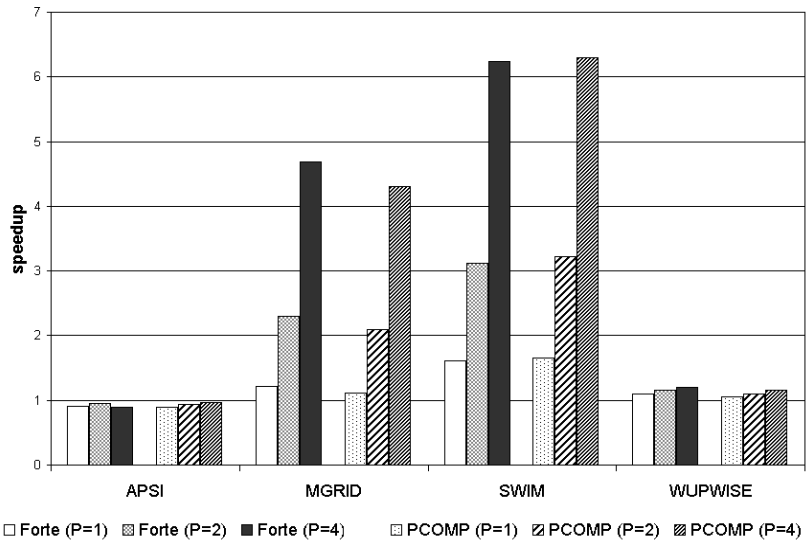

Fig. 4. Automatic generation of OpenMP programs. Speedup of benchmarks as executed on SUN Enterprise 4000 using our OpenMP postpass. The OpenMP codes are parallelized by the Polaris parallelizer with our OpenMP postpass. The codes are compiled by the SUN Forte compiler, and by our PCOMP Portable OpenMP translator with a sequential compiler. `P` represents the number of processors used.

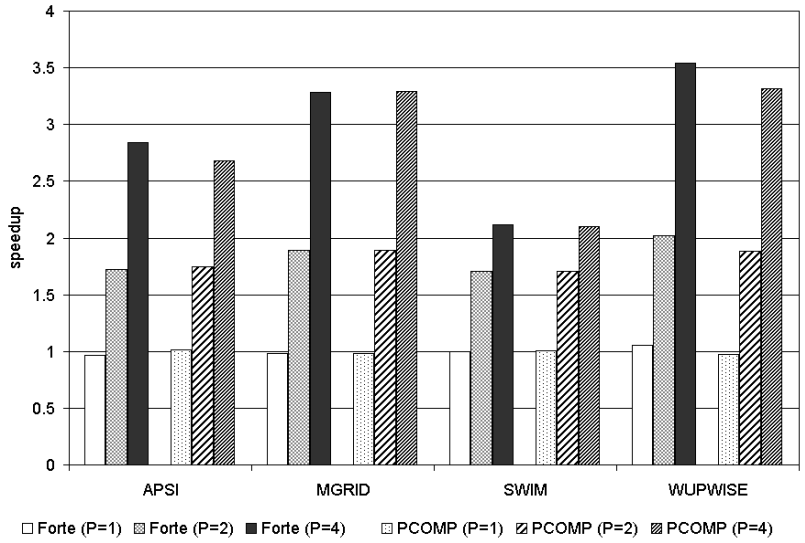

Fig. 5. Compilation of OpenMP programs. Speedup of benchmarks as executed on SUN Enterprise 4000. The OpenMP Suite codes are compiled by the SUN Forte compiler and our PCOMP Portable OpenMP compiler. `P` represents the number of processors used.

References

1. William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.
2. Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP compiler for a SMP cluster. In *The 1st European Workshop on OpenMP (EWOMP'99)*, pages 32–39, September 1999.
3. C. Brunschen and M. Brorsson. OdinMP/CCp - a porable implementation of OpenMP for C. *Concurrency: Practice and Experience*, (12):1193–1203, 2000.
4. Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. Performance analysis of parallel compiler backends on shared-memory multiprocessors. In *Compilers for Parallel Computers (CPC2000)*, pages 305–320, January 2000.
5. Mike Voss. Portable level-parallelism for shared-memory multiprocessor architectures. Master's thesis, Electrical and Computer Engineering, Purdue University, December 1997.
6. Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 444–448, 95.
7. John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.
8. Sun Microsystems Inc., Mountain View, CA, http://www.sun.com/servers/enterprise/e4000/index.html. *Sun Enterprise 4000*.