A Source-to-Source OpenMP Compiler

Mario Soukup and Tarek S. Abdelrahman
The Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{soukup,tsa}@eecg.toronto.edu

Abstract

In this paper we describe an implementation of the Fortran OpenMP standard for compiler-directed shared-memory parallel programming. The implementation is source-to-source, i.e., it takes as input a Fortran program with OpenMP directives and produces a Fortran program which explicitly creates and manages parallelism according to the specified directives. This makes the implementation portable, only requiring a standard POSIX thread library to create and synchronize threads. Experimental evaluation of the implementation indicates that it has low run-time overhead, making it competitive with commercial implementations. Furthermore, the obtained results show linear speedups for sample benchmark applications.

1. Introduction

The OpenMP standard [3] for compiler-directed shared-memory parallel programming has gained considerable acceptance in recent years. The standard specifies a number of compiler directives which a programmer may insert in a sequential program to indicate how the program is to be parallelized on a shared-memory multiprocessor. An OpenMP compiler converts the sequential program into a parallel one according to these directives. This approach to parallel programming has the advantage of relieving the programmer from the mundane tasks of parallel programming, such as the creation, synchronization and management of threads. There are two versions of the OpenMP standard, one for the Fortran language and one for C/C++.

Today, there exists a number of commercial compilers that support OpenMP. They include the MIPSpro compiler from SGI [8], the F95 Fortran compiler from Sun [10] and Guide F90 from Kuck and Associates [6]. Although these compilers provide an efficient implementation of the standard, the source code of the OpenMP implementation is not available to compiler researchers, making it difficult to use in a research environment. Further, these implementation are specific to the platforms they target, making them non-portable. In response, there has been a number of public domain OpenMP implementation; they include OdinMP/CCp [4], Omni OpenMP [7], and NanosCompiler [1]. Unfortunately, these implementation are for the C/C++ version of the standard, and/or are specific to a particular platform. There does not appear to be a portable public-domain implementation of the Fortran version.

In this paper, we describe a portable implementation of the OpenMP 1.0 standard for the Fortran language. The implementation is source-to-source, meaning that it converts the input program, in Fortran source, which contains OpenMP directives into an output program, also in Fortran source. The output program explicitly creates and synchronizes parallel threads to implement parallelism, as indicated by the OpenMP directives in the input program. The implementation requires standard POSIX threads to create, manage and synchronize threads. The use of source-to-source translation and run-time support that is standard on most systems makes our implementation and the parallel code it generates portable across platforms. Our experimental evaluation of the implementation indicates that it incurs little overhead at run-time and that linear speedup can be achieved using it on sample benchmarks applications.

The remainder of this paper is organized as follows. A brief overview of OpenMP is given in Section 2. The design and implementation of our compiler are described in Section 3. Experimental evaluation of the compiler is presented in Section 4. Related work is reviewed in Section 5. Finally, concluding remarks are given in Section 6.

2. An overview of OpenMP

OpenMP [3] is a standard that specifies a set of compiler directives which a programmer may insert in a sequential program. The programmer inserts these directives to describe the intended parallel execution of the program. An OpenMP compiler then uses the directives to parallelize the program.

The example shown in Figure 1 illustrates how some OpenMP directives may be used to describe parallelism in a short Fortran program. The directives are indicated as comments starting with the string `C$OMP`. The `PARALLEL` directive indicates that the block of code that follows, up to the `END PARALLEL` directive, is to be executed by more than one thread. The directive has a `SHARED` clause, which identifies variables that are shared by the threads during parallel execution; `j` in the case of the example. The `DO` directive specifies that the iterations of the DO loop that immediately follows are to be executed in parallel. The `SCHEDULE(STATIC)` clause indicates that the iterations of the DO loop are to be divided equally among the threads in contiguous blocks at compile time. The `PRIVATE` clause of the `DO` directive indicates that the variable `i` is to be treated as private to each thread; i.e., each thread has a local copy of the variable.

Hence, in the above example, the programmer specifies through the directives which segments of the program are to be executed in parallel; how the work is to be divided

```
        program test
        integer*8 i,j
        j = 0
        do i=1, 10000000, 1
           j = j + 1
        end do
        print *, i, j
        end
```

(a) Example program

```
        program test
        integer*8 i,j
        j = 0
c$OMP   PARALLEL SHARED(j)
c$OMP   DO SCHEDULE(STATIC) PRIVATE(i)
        do i=1, 10000000, 1
           j = j + 1
        end do
c$OMP   END PARALLEL
        print *, i, j
        end
```

(b) Program with OpenMP directives

Figure 1. An example of the use of OpenMP directives.

among the processors; and which variables are shared and which are private.

The main directives of OpenMP for Fortran are described in the remainder of this section. The reader is referred to the specification [3] for a complete description.

2.1 The PARALLEL directive

The PARALLEL and END PARALLEL directives define a parallel region. A parallel region is a block of code that is to be executed by a team of multiple threads in parallel. The team has a designated thread called the team *master*. This is the fundamental parallel construct in OpenMP that starts parallel execution. This directive has the following format:

```
C$OMP PARALLEL [clause [[,] clause ]...]
block
C$OMP END PARALLEL
```

The PARALLEL directive has seven optional clauses that each take one or more arguments. The clauses are PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION, IF, and COPYIN.

The PRIVATE clause declares a list of variables to be private to each thread in the team. The SHARED clause declares a list of variables to be shared among all the threads in the team; all threads within a team access the same storage area for these shared variables. The DEFAULT clause allows the user to specify a default PRIVATE, SHARED, or NONE scope attribute for all variables in the lexical extent of any parallel region. The FIRSTPRIVATE clause declares a list of variables that are private. However, private copies of these variables are initialized with the values of the corresponding variables before the PARALLEL region. The REDUCTION clause indicates a reduction on the variables that appear in list, with an operator or an intrinsic. The COPYIN clause applies only to common blocks that are declared as THREADPRIVATE. A COPYIN clause on a parallel region species that the data in the master thread of the team be copied to the thread private copies of the common block at the beginning of the parallel region. When the IF clause is present in a parallel region, the enclosed code region is executed in parallel only if a scalar logical expression evaluates to true. Otherwise, the parallel region is serialized.

2.2 The DO directive

The DO directive specifies that the iterations of the immediately following DO loop must be executed in parallel. The iterations of the loop are distributed across threads that already exist, i.e., ones created by a preceding PARALLEL directive. The format of this directive is as follows:

```
C$OMP DO [ clause [ [,] clause ] ... ]
do_loop
[ C$OMP END DO [ NOWAIT ] ]
```

The DO directive has six optional clauses that take one or more arguments: PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, SCHEDULE, and ORDERED. The PRIVATE, FIRSTPRIVATE, and REDUCTION clauses of this directive carry similar syntax and semantics to their respective counterparts of the PARALLEL directive. The LASTPRIVATE clause has similar semantics to the FIRSTPRIVATE clause of the PARALLEL directive, but the thread that executes the sequentially last iteration of the loop updates corresponding variables outside the DO loop. The SCHEDULE clause specifies how iterations of the DO loop are divided among the threads of the team. There are four different types of scheduling schemes:

- STATIC: the iterations of the loop are divided into pieces (chunks) whose size (chunk size) may specified in the clause. If the chunk size is not specified, the iterations are divided equally in contiguous chunks among the threads.
- DYNAMIC: the iterations of the loop are divided into chunks, as above. However, the assignment of chunks to threads is done at run-time. When a thread finishes a chunk of the iteration space, it dynamically obtains the next chunk of iterations.
- GUIDED: the iterations of the loop are divided into chunks and assigned to threads at run-time as above. However, the size of the chunks is not constant. The chunk size is reduced exponentially as the chunks are assigned to threads using the following rule: chunk size = number of iterations left / number of threads.
- RUNTIME: the specification of the scheme of scheduling is deferred until run time, and is determined by examining the OMP_SCHEDULE environment variable.

The NOWAIT option in the END DO segment of the directive allows parallel threads to proceed past the end of the DO loop without waiting for other threads that are still executing iterations of the loop.

2.2.1 The SECTIONS directive

The SECTIONS directive specifies that the enclosed sections of code are to be divided among threads in the team and that each section is executed once by one thread. The format of this directive is as follows:

```
C$OMP SECTIONS [clause [[,] clause ]...]
[ C$OMP SECTION ]
block
[ C$OMP SECTION
block ]
C$OMP END SECTIONS [ NOWAIT ]
```

The optional clauses (PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION) are similar to the respective clauses of the PARALLEL and DO directives.

3. Design and Implementation

In this section, we describe the design and implementation of our OpenMP compiler. First, we describe the front end of the compiler, then we describe how the various constructs of OpenMP are implemented, and finally we describe the run-time support.

3.1 Front end

We used the Polaris compiler [2] from the University of Illinois at Urbana-Champaign to implement the front end of our OpenMP compiler. Although Polaris includes facilities to detect parallelism in sequential programs and generates parallel code, we did not use these facilities. Rather, we used the core of the Polaris compiler to parse, represent and manipulate Fortran code. This allowed us to add code to parse of OpenMP directives, to restructure the input program to package parallel regions into threads, and to insert calls to the run-time system to create and schedule threads.

3.2 Directive implementation

In this section, we describe how our compiler implements the main OpenMP directives. A description of the implementation of all directives is not included here, and may be found in [9].

3.2.1 The PARALLEL directive

The body of the PARALLEL directive is extracted from the input program and is packaged as a new stand-alone subroutine in the output program. In place of the extracted block of code, a call is made to a run-time routine, which takes the address of the new subroutine as an argument. The run-time routine creates multiple threads that then execute the new subroutine. This transformation is illustrated in Figure 2 for a sample program.

```
      program test
      print *, "Started"
c$OMP PARALLEL
      print *, "Hello world"
c$OMP END
      print *, "Finished"
      end
```

(a) Input program

```
SUBROUTINE omp_sub1(omp_thr_num)
INTEGER*4 omp_thr_num
PRINT *, 'Hello world'
END

PROGRAM test
EXTERNAL f_pthread_create_threads
EXTERNAL omp_sub1
PRINT *, 'Started'
CALL f_pthread_create_threads(omp_sub1,0)
PRINT *, 'Finished'
STOP
END
```

(b) Output program

Figure 2. Example implementation of PARALLEL.

All variables designated as private in the PRIVATE clause of the PARALLEL directive are simply declared within the new subroutine. Shared variables, on the other hand, are moved to a common block created by the compiler to allow shared access to them across subroutines. Variables designated in a FIRSTPRIVATE clause are declared in the new subroutine and initialized to the values of the original variables. These values are obtained via common block temporary variables that take on the values of the original variables. Figure 3 show an example of the code transformations involving PRIVATE, SHARED, and FIRSTPRIVATE variables.

When an IF clause is used in a PARALLEL directive, two different run-time calls are inserted into the original code; one that creates multiple threads, and another that creates only one thread. Both of these calls are mutually exclusive, and need to be guarded by an if construct that tests the value of scalar logical expression specified in the IF clause.

The implementation of the REDUCTION clause is slightly more complicated. All variables that appear in a reduction clause must be shared in the enclosing context. A private copy of each of these variables is created in the new subroutine and an appropriate default initialization statement is inserted at its beginning. A new variable is defined and aliased to represent the original reduction variable through a common block. This new variable is updated at the end of the parallel region to reflect the result of the reduction. To prevent multiple threads accessing the variable at the same time, calls to run-time synchronization routines are inserted before and after the update: set lock, and unset lock. An example of the implementation of PARALLEL with a REDUCTION clause is shown in Figure 4.

3.2.2 The DO directive

The PRIVATE, FIRSTPRIVATE, and REDUCTION clauses of this directive are handled in a similar manner to the PARALLEL directive. Hence, we focus on the handling

```
      program test
      integer i,j,k
      i = 1
      j = 2
      k = 3
      print *, "Started"
c$OMP PARALLEL PRIVATE(i)
c$OMP+ SHARED(j) FIRSTPRIVATE(k)
      print *, i,j,k
c$OMP END PARALLEL
      print *, "Finished"
      end
```

(a) Input program

```
SUBROUTINE omp_sub1(omp_thr_num)
INTEGER*4 i, j, k
INTEGER*4 omp_init1, omp_thr_num
COMMON /omp_cb2/ omp_init1
COMMON /omp_cb1/ j
k = omp_init1
PRINT *, i, j, k
END

PROGRAM test
EXTERNAL f_pthread_create_threads
EXTERNAL omp_sub1
INTEGER*4 i, j, k
COMMON /omp_cb2/ k
COMMON /omp_cb1/ j
i = 1
j = 2
k = 3
PRINT *, 'Started'
CALL f_pthread_create_threads(omp_sub1,0)
PRINT *, 'Finished'
STOP
END
```

(b) Output program

Figure 3. Example implementation of `PARALLEL` with private and shared variables.

of the `SCHEDULE` clause of the directive. There are four different types of scheduling schemes: `STATIC`, `DYNAMIC`, `GUIDED`, and `RUNTIME`. Their handling by the compiler is described below.

- `STATIC`. To implement this scheduling scheme, two temporary variables are created to hold the loop bounds for each thread. Code to compute the values of these variables is inserted before the loop in the output program. Figures 5 and 6 respectively illustrate the implementation for a sample program when the `SCHEDULE` clause does and does not specify a chunk size. The `OMP_GET_NUM_THREADS()` and `OMP_GET_THREAD_NUM()` are run-time routines that return the number of threads and the calling thread ID respectively. When the chunk size is specified, an additional loop is created around the loop that is to be scheduled. This is necessary since there might be multiple chunks of iterations that each thread will work on.

```
      program test
      integer i
      print *, "Started"
c$OMP PARALLEL REDUCTION(+:i)
      print *, "Hello world"
      i = i+1
c$OMP END PARALLEL
      print *, "Finished"
      end
```

(a) Input program

```
SUBROUTINE omp_sub1(omp_thr_num)
EXTERNAL omp_set_lock, omp_unset_lock
INTEGER*4 i, omp_mutex1
INTEGER*4 omp_reduce2, omp_thr_num
COMMON /omp_cb2/ omp_reduce2
COMMON /omp_cb1/ omp_mutex1
i = 0
PRINT *, 'Hello world'
i = i+1
CALL omp_set_lock(omp_mutex1)
omp_reduce2 = omp_reduce2+i
CALL omp_unset_lock(omp_mutex1)
END

PROGRAM test
EXTERNAL f_pthread_create_threads
EXTERNAL omp_init_lock, omp_sub1
INTEGER*4 i, omp_mutex1
COMMON /omp_cb2/ i
COMMON /omp_cb1/ omp_mutex1
PRINT *, 'Started'
CALL omp_init_lock(omp_mutex1)
CALL f_pthread_create_threads(omp_sub1,0)
PRINT *, 'Finished'
STOP
END
```

(b) Output program

Figure 4. Example implementation of `PARALLEL` with reduction variables.

- `DYNAMIC`. An example of the implementation of this scheme is shown in Figure 7. The call to `F_OMP_INIT_SCHED` allows the run-time system to record the type of scheduling, and the initial scheduling data. The subsequent call to `F_OMP_GET_SCHED_VARS` function calculates the next set of iterations a given thread executes.

- `GUIDED`. The implementation of this scheme is similar to that of `DYNAMIC`. The only difference is that the chunk size is computed differently, as was explained earlier.

- `RUNTIME`. The implementation of this scheme is also similar to that of `DYNAMIC`. However, the actual scheme of scheduling is set when the initialization

```
      DO i=init, limit, step
         code
      ENDDO
```

(a) Input code

```
num_of_threads = OMP_GET_NUM_THREADS()
thread_num = OMP_GET_THREAD_NUM()
DO ii=0,(limit-init)/
        (chunk*step*num_of_threads), 1
   new_init = init + (thread_num+ii*
             num_of_threads)*chunk*step
   new_limit = new_init+(chunk-1)*step
   IF ( limit/new_limit .EQ. 0 ) THEN
      new_limit = limit
   ENDIF
   DO i=new_init, new_limit, step
      code
   ENDDO
ENDDO
```

(b) Output code

Figure 5. Example implementation of STATIC when the chunk size is specified.

```
      DO i=init, limit, step
         code
      ENDDO
```

(a) Input code

```
num_of_threads = OMP_GET_NUM_THREADS()
thread_num = OMP_GET_THREAD_NUM()
size = ((limit-init)/step+1 )/
        num_of_threads
IF (size*num_of_threads .NE.
         (limit-init)/step+1) THEN
   size = size + 1
ENDIF
new_init = init + thread_num*size*step
new_limit = new_init+(size-1)*step
IF ( limit/new_limit .EQ. 0 ) THEN
   new_limit = limit
ENDIF
DO i=new_init, new_limit, step
   code
ENDDO
```

(b) Output code

Figure 6. Example implementation of STATIC when the chunk size is not specified.

```
      DO i=init, limit, step
         code
      ENDDO
```

(a) Input code

```
  thread_num = OMP_GET_THREAD_NUM()
  CALL F_OMP_INIT_SCHED(
       SCHEDULING_TYPE,init,limit,step,
       chunk, omp_thr_num )
  new_init = init
  new_limit = limit
  new_step = step
1 CONTINUE
  IF ( F_OMP_GET_SCHED_VARS(
       new_init, new_limit, thread_num )
               .EQ. 0 ) THEN
     GOTO 2
  ENDIF
  DO i=new_init, new_limit, new_step
     code
  ENDDO
  GOTO 1
  2 CONTINUE
```

(b) Output code

Figure 7. Example implementation of DYNAMIC scheme.

function F_OMP_INIT_SCHED is called. This function checks the OMP environment variables, and sets up all necessary information about scheduling at that time. The iterations are then, obtained using the mechanisms described above.

3.2.3 The SECTIONS directive

To ensure sections are executed by only one thread, an if statement is inserted around every section in the block. The if guard checks that the number of the thread that should execute a section corresponds to the number of the thread currently executing the code. If there was no match, the section is not executed by the thread. All of the optional directive clauses (PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION) are handled in a manner similar to how they would be handled in a PARALLEL or DO directive case.

3.3 Run-time support

A run-time library provides support for our compiler-generated programs. The library contains routines that are invoked from the Fortran program. The routines themselves are written in C and implement their required behaviour through extensive usage of a POSIX thread library. The routine calls are inserted by the source-to-source compiler phase, and provide mechanisms for the creation, initialization and synchronization of threads. In addition, the run-time library maintains an internal storage of program specific conditions, and states, that the run-time routines use.

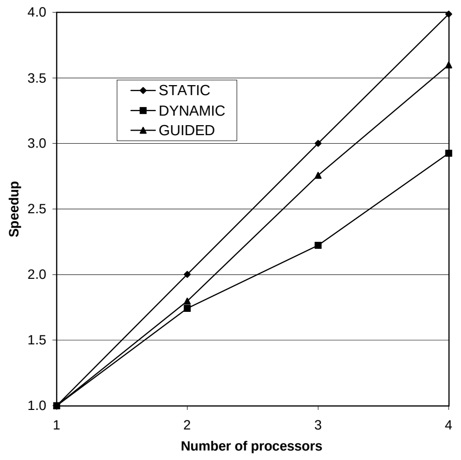

Figure 8. Speedup of the synthetic benchmarks.

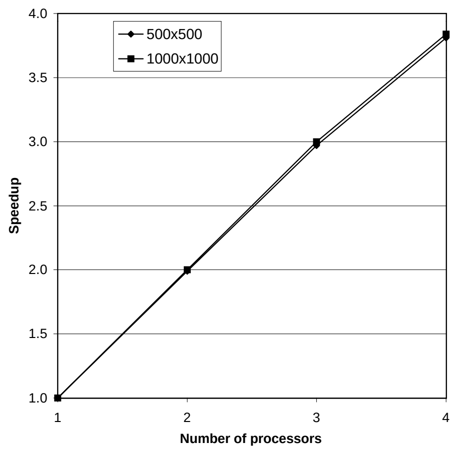

Figure 9. Speedup of the Jacobi application.

Finally, the runtime provides routines to handle environment variables.

4. Experimental Evaluation

We used our compiler to generate parallel code for a number of standard benchmarks. We executed the resulting parallel code on a 4-processor (400 Mhz) Sun Ultra Enterprise 450-BA model 440 multiprocessor. This allowed us to measure both the overhead incurred by our implementation of various `OpenMP` constructs and to measure the speedup of applications parallelized using our `OpenMP` compiler.

We first give speedup (the ratio of the execution time of the sequential program, with no `OpenMP` directives, to the execution time of the parallel program) results for two `OpenMP` programs. The first is a synthetic benchmark that consists of two nested loops that perform simple computations. The outer loop is executed in parallel using the `PARALLEL DO` directive and scheduled in three different ways: `STATIC`, `DYNAMIC` and `GUIDED`. Hence, this benchmark provides and upper bound on the speedup that may be obtained. The second program is the `Jacobi` benchmark available at the `OpenMP` web site (www.openmp.org). In this benchmark, matrix sizes of 500×500 and 1000×1000 are used along with the `STATIC` scheduling scheme.

Figures 8, and 9 show the speedup of the two programs as a function of the number of processors. The figures indicate that both benchmarks speedup well.

Second, we give measurements of the overhead incurred by the run-time library. The Edinburgh Parallel Computing Centre (EPCC) developed a set of micro-benchmarks to measure run-time overhead incurred by various `OpenMP` constructs [5]. In particular, these micro-benchmarks measure the overhead of barrier synchronization, mutual exclusion, and loop scheduling. The EPCC publishes overhead data for a number of commercial `OpenMP` compilers, which we use to compare to the overhead of our own implementation.

Figure 10 shows the overhead (in processor cycles[1]) of barrier synchronization, which results from the use of

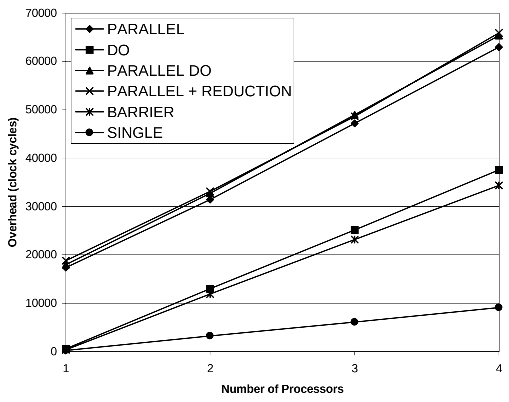

Figure 10. Barrier overhead in processor cycles.

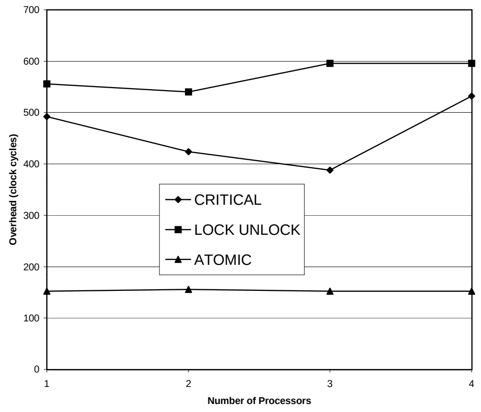

Figure 11. Mutual exclusion overhead in processor cycles.

[1]This allows comparisons to other machines with the same processor architecture, but different clock speeds.

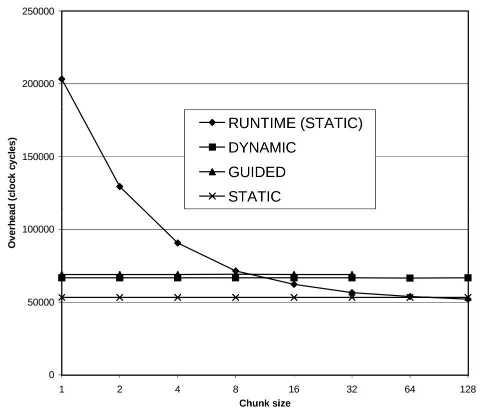

Figure 12. Loop scheduling overhead in processor cycles.

PARALLEL, DO, PARALLEL DO, BARRIER, and SINGLE directives. The figure indicates that although the amount of overhead associated with each of the above constructs is different, the overhead increases linearly with the number of processors. This is because we use the underlying implementation of the POSIX library which does not provide scalable barrier synchronization. Nonetheless, the amount of overhead is small.

Figure 11 shows the overhead of mutual exclusion, which results from the use of CRITICAL, ATOMIC, LOCK, and UNLOCK directives. The figure indicates that the overhead is small and that it does not increase with the number of processors.

Figure 12 shows the overhead of loop scheduling, which results from the use of the various scheduling options of a DO construct at various chunk sizes. The figure indicates that the overhead is small for the schemes. In the case of the RUNTIME scheme, STATIC scheduling was used. The overhead is high for small chunk sizes, but declines as the chunk size is increased.

We compare the overhead of our implementation with the overhead of a commercial OpenMP compiler: version 3.7 of Guide F90 by Kuck and Associates. The results of running the EPCC micro-benchmarks using this commercial compiler on a 400-Mhz Sun E3500 are reported on the EPCC web site (www.epcc.ed.ac.uk) in processor cycles. Since the clock rate for this machine is identical to the one we use, it is possible to make a meaningful comparison. The overhead of barrier synchronization for our compiler is about 2 to 3 times higher than that for KAI's compiler. However, the overhead of mutual exclusion is smaller by a factor of 2 and the overhead of loop scheduling is significantly smaller (by a factor of 10 for some chunk sizes). Hence, in general, the run-time overhead of our compiler is of the same order as that of this commercial compiler.

5. Related work

Brunschen and Brorsson [4] describe the implementation of OdinMP/CCp, a portable implementation of OpenMP for C. Their implementation is also source-to-source and uses POSIX threads to implement parallelism. Hence, our compiler shares similarities to theirs. However, our compiler targets Fortran instead of C.

Ayguade et al. [1] describe a research implementation of OpenMP centered around the hierarchical intermediate representation of Parafrase II compiler. Their goal is to provide an infrastructure for exploring extensions of OpenMP. They target the Illinois-Intel multithreading library, which restricts the portability of their implementation. In contrast, we use POSIX and, hence, gain portability.

Sato et al. [7] present the design of an OpenMP implementation for C on SMP clusters. However, they rely on a C++ run-time library specific to their SMP cluster, restricting portability.

6. Concluding remarks

In this paper, we described the design and implementation of a Fortran source-to-source OpenMP compiler. The compiler is portable and requires only standard POSIX threads support on a target system. Experimental evaluation of parallel code generated by our compiler indicates that it has little overhead and that it can deliver linear speedup on sample applications.

We plan to release our implementation in public domain. We also plan to use it to carry out research on OpenMP, particularly on the incorporation of data placement directives.

References

[1] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. NanosCompiler: a research platform for OpenMp extensions. In *European Workshop on OpenMP*, 1999.

[2] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[3] OpenMP Architecture Review Board. OpenMP: a proposed standard API for shared memory programming. *http://www.openmp.org*, 1997.

[4] C. Brunschen and M. Brorsson. OdinMP/CCp - a portable implementation of OpenMp for C. In *European Workshop on OpenMP*, 1999.

[5] M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *European Workshop on OpenMP*, 1999.

[6] Kuck and Associates, Inc. KAP/Pro Toolset, http://www.kai.com.

[7] M. Sate, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMp compiler for an SMP cluster. In *European Workshop on OpenMP*, 1999.

[8] Silicon Graphics, Inc. The Mips/Pro Compiler, http://www.sgi.com.

[9] M. Soukup. A source-to-source openmp compiler. Master's thesis, University of Toronto, Toronto, Ontario, Canada, In progress, 2001.

[10] Sun Microsystems, Inc. The F95 Sun Compiler, http://www.sun.com.