A Source-to-Source OpenMP Compiler

by

MARIO SOUKUP

A Thesis submitted to fulfil the equivalent of three half courses
of the requirements for the degree of
Master of Engineering
in the

Department of Electrical and Computer Engineering
University of Toronto

August 2001

Abstract

This thesis describes 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 an input a FORTRAN program with OpenMP directives and produces a FORTRAN program, which explicitly creates and manages parallelism according to the specified directives. This approach makes the code generated by 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.

TABLE OF CONTENTS

Chapter 1

Introduction

The high-end computer market has undergone tremendous growth in the past decade. The technological advancements pushed a uniprocessor environment to new levels that clearly resemble the parallel computing paradigm. Modern superscalar processors function internally as multiple separate units capable of executing multiple instructions at the same time. Their future designs propose even greater internal separation eventually leading to what some vendors are already planning: fully functional multiple processors within one chip. These seemingly new ideas and advancements have been around for a number of years. Their biggest (commercial) obstacle was the difficulty of programming explicitly in a parallel environment. To alleviate this difficulty of parallel programming, some compiler vendors started developing new parallel languages, while others extended the functionality of the existing programming languages via parallel directives. As always the case when many vendors compete in the same field, it took a long time for a successful, shared-memory parallel programming directive standard to emerge.

In 1997, a parallelizing directive standard called OpenMP [14] emerged. The standard specified 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. OpenMP standard provides specifications for Application Program Interface (API) for FORTRAN, C, and C++.

Today, there exist a number of commercial compilers that support OpenMP. They include the MIPSpro compiler from SGI [15], the F95 FORTRAN compiler from Sun [16] and Guide F90 from Kuck and Associates [10]. 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. Furthermore, these

implementations are specific to the various platforms they target, making them non-portable. In response, there has been a number of public domain OpenMP implementation; they include OdinMP/CCp [5], Omni OpenMP [13], and NanosCompiler [2]. Unfortunately, these implementations are either implementations of the C version of the standard, or are specific to a particular platform. There does not appear to be a portable public-domain implementation of the FORTRAN version.

The goal of this thesis is to implement a portable, public domain, OpenMP v1.0 FORTRAN compiler. The compiler is designed to be a source-to-source compiler that transforms OpenMP-enhanced FORTRAN code into pure FORTRAN that uses POSIX thread routines to create parallelism as specified by the OpenMP directives.

The benefits of an OpenMP source-to-source compiler are many. The source-to-source design allows for the portability of the generated code across different platforms. The compiler provides FORTRAN users with OpenMP programming capability at no cost. The programming environment changes required by the compiler are minimal, and most importantly, existing development tools such as debuggers, and profilers, do not have to be replaced. The compiler will be an open source (i.e. public domain) application that is open for improvements and changes by others, and hence, it could be used for further research purposes.

1.1 Thesis Objective

The objective of this thesis is to create a portable implementation of the OpenMP 1.0 standard for the FORTRAN language. The implementation is to be source-to-source, meaning that it converts an input program, in FORTRAN source, which contains OpenMP directives into an output program, also in FORTRAN source. The output program will explicitly create and synchronize parallel threads to implement parallelism, as indicated by the OpenMP directives in the input program. The implementation will require standard POSIX threads to create, manage and synchronize threads. The use of a source-to-source translator, and run-time support that is available on most systems, will make the implementation portable across platforms. Figure 1 shows the intended process of creating and executing OpenMP programs using the source-to-source compiler implementation.

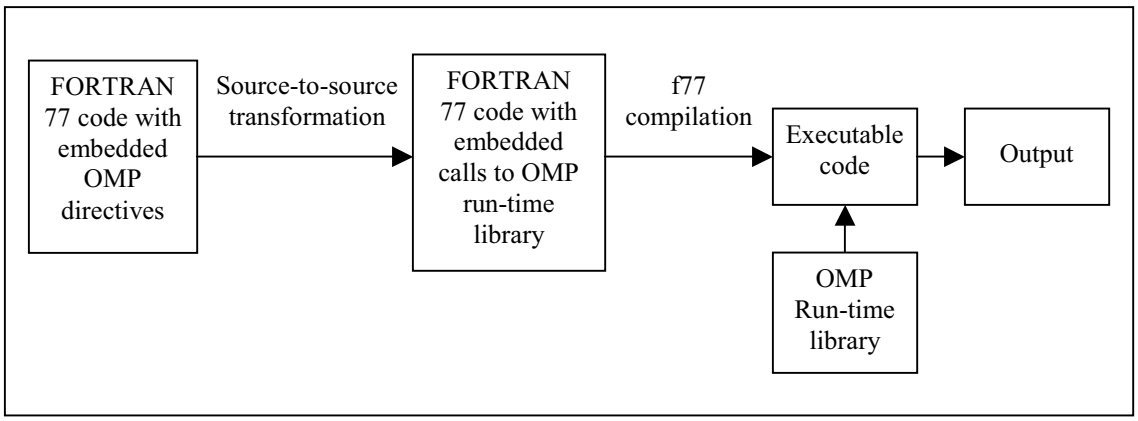

Figure 1 OpenMP source-to-source compiler usage diagram.

More specifically, the source-to-source implementation has the following features:

1. It acts as a FORTRAN pre-processor that takes as an input FORTRAN 77 code embedded with OpenMP directives. The processed output is pure FORTRAN 77 code interlaced with calls to a portable OpenMP run-time library.
2. The processed FORTRAN code is required to be semantically identical to the original code, and be able to run in parallel according to the Open MP API 1.0[1] directive specifications placed in the original code.
3. Run-time library will provide the necessary support for the parallelization implementation through the usage of POSIX pthreads. This will be the only system requirement.

The limitations of the current implementation are: the front end implements no semantic error checking, and the FLUSH, and THREADPRIVATE directives, have not been implemented.

1.2 Thesis Organization

The remainder of this thesis is organized in the following manner. Chapter 2 provides background information on the OpenMP API specification, the POLARIS compiler used to implement the front end of our compiler, and the POSIX standard. Chapter 3 outlines the design and implementation of the source-to-source OpenMP compiler. It describes the algorithms used and gives concrete examples of the compiler behaviour. Chapter 4 looks at the compiler performance through various benchmarks, and provides an analysis and evaluation of the results.

[1] Open MP API v2.0 was released in November 2000, while the thesis implementation was underway.

The chapter also contains a detailed description of the testing procedures used to ensure the correctness of the generated code. Chapter 5 reviews, and compares the work done in three related projects with the design and implementation of the OpenMP compiler. Chapter 6 concludes the thesis and provides some direction of the future work. Finally, Appendices A, and B contain the benchmark code and sample test programs created for this thesis.

Chapter 2

Background

This chapter presents background material that is related to the implementation of the OpenMP compiler. Section 2.1 provides the history of the OpenMP standard, and describes its API and runtime support. Section 2.2 gives an overview of the POSIX standard, and describes the POSIX interfaces that are used in this thesis. Finally, section 2.3 describes the structure and functionality of the POLARIS compiler, which is used to perform the source-to-source restructuring of input programs.

2.1 OpenMP

In the early 90's, vendors of shared-memory machines started supplying similar, directive-based, FORTRAN programming extensions. These directives were intended to be used as an augmentation of a serial FORTRAN program stating to the compiler which loops in the program are to be parallelized. The compiler then became responsible for all of the work required in parallelization of these loops across multiple processors.

Most of the competing directive extensions introduced were functionally similar, but as always is the case with similar products from different companies, they were diverging. The first real attempt at a standard was the ANSI X3H5 draft [1] in 1994. It was never adopted, largely due to waning interest as distributed memory machines became more popular.

In the spring of 1997, as newer shared memory machine architectures started to become prevalent, OpenMP standard specification was created taking over where ANSI X3H5 had left off. Since that time, three versions have been released:

1. OpenMP FORTRAN API version 1.0 released October 28, 1997
2. OpenMP C/C++ API version 1.0 released in late 1998

3. OpenMP FORTRAN API version 2.0 released in November, 2000

In order to illustrate the OpenMP standard, a sample FORTRAN program is used to show the parallelization methodology of the OpenMP directives.

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

Figure 2. Original FORTRAN code.

```
 1:       program test
 2:       integer*8 i,j
 3:       j = 0
 4: c$OMP PARALLEL SHARED(j)
 5: c$OMP DO SCHEDULE(STATIC,100) PRIVATE(i)
 6:       do i=1, 10000000, 1
 7:          j = j + 1
 8:       end do
 9: c$OMP END PARALLEL
10:       print *, i, j
11:       end
```

Figure 3. FORTRAN code with embedded OpenMP directives**.**

Figure 3 shows three additional "comments" that were inserted into the original code shown in Figure 2. Line 4 in the code contains a PARALLEL OpenMP directive. This directive informs the compiler that the block of code that follows the directive has to be executed by multiple threads. The end of the parallel region is signified by the END PARALLEL directive on line 9.

The PARALLEL directive also allows for some optional clauses. One such clause is the SHARED clause used on line 4. It instructs the compiler that all variables listed as arguments of the clause, are to be shared across all threads running the parallel region. For example, variable j in Figure 3 is a SHARED variable.

Line 5 in the code contains the DO directive that specifies that the iterations following the do loop are to be executed in parallel. Its SCHEDULE clause specifies how the execution of iterations is to be divided among the threads. In the example in Figure 3, the STATIC scheduling

will divide the iterations into pieces of size 100. The pieces are then statically assigned to threads in the team in a round-robin fashion.

The following sections describe in detail the syntax and functionality of the OpenMP directives, environment routines, and environment variables.

2.1.1 OpenMP Directives

The OpenMP directives are special FORTRAN comments that are identified with a unique sentinel. Their format is as follows:

```
sentinel directive_name [ clause [ [,] clause ] ... ]
```

The following sections describe the type and functionality of the OpenMP directives.

2.1.1.1 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 multiple threads in parallel. This is the fundamental parallel construct in OpenMP that starts parallel execution. These directives have the following format:

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

The PARALLEL directive has seven optional clauses that take one or more arguments: PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION, IF, and COPYIN. The purpose of these clauses is to indicate the manner in which variables are used inside the parallel region. 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 the 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 performs 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 used in a parallel region specifies that the common block data in the master thread of the team, is to be copied to the thread private copies of the common block at the beginning of the parallel region. When an IF clause is present in a parallel region, the enclosed code region is executed in parallel only if the scalar logical expression specified in the clause evaluates to true. Otherwise, the parallel region is serialized.

For more details on directive clauses, see section 2.1.2.

2.1.1.2 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 DO loop are distributed across threads that have already been created by a PARALLEL construct. 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 in the PARALLEL directive. The LASTPRIVATE clause has similar semantics to the FIRSTPRIVATE clause of the PARALLEL directive, except that the thread that executes the sequentially last iteration of the loop, must update the versions of the objects it had before the DO construct. The SCHEDULE clause specifies how iterations of the DO loop are to be divided among the threads in the team. There are four different types of scheduling schemes: STATIC, DYNAMIC, GUIDED, and RUNTIME. 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 the other threads that are still executing iterations of the loop. For more details on directive clauses, see section 2.1.2.

2.1.1.3 The SECTIONS directive

The SECTIONS directive specifies that the enclosed sections of the code are to be divided among threads in the team. Each section is executed once by a thread in the team. 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 SECTIONS directive has four optional clauses that take one or more arguments: PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION. For more details on directive clauses, see section 2.1.2.

2.1.1.4 The SINGLE directive

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team. Threads in the team that are not executing the SINGLE directive wait at the END SINGLE directive unless NOWAIT is specified. The format of this directive is as follows:

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

The SINGLE directive has two optional clauses that take one or more arguments: PRIVATE, and FIRSTPRIVATE. For more details on directive clauses, see section 2.1.2.

2.1.1.5 The PARALLEL DO directive

The PARALLEL DO directive provides a shortcut form for specifying a parallel region that contains a single DO directive. The format of this directive is as follows:

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

The PARALLEL DO directive clauses can be one of the clauses accepted by the PARALLEL and DO directives. For more details on directive clauses, see section 2.1.2.

2.1.1.6 The PARALLEL SECTIONS directive

The PARALLEL SECTIONS directive provides a shortcut form for specifying a parallel region that contains a single SECTIONS directive. The format of this directive is as follows:

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

The PARALLEL SECTIONS directive clauses can be one of the clauses accepted by the PARALLEL and SECTIONS directives. For more details on directive clauses, see section 2.1.2.

2.1.1.7 The MASTER directive

The code enclosed within the MASTER directive block is executed by the master thread of the team. The directive has the following format:

```
C$OMP MASTER
block
C$OMP END MASTER
```

OMP MASTER directive has no clauses.

2.1.1.8 The CRITICAL directive

The CRITICAL directive restricts access to the enclosed code to only one thread at a time. These directives have the following format:

```
C$OMP CRITICAL [ ( name ) ]
block
```

```
C$OMP END CRITICAL [ ( name ) ]
```

The CRITICAL directive has no clauses.

2.1.1.9 The BARRIER directive

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. The format of this directive is as follows:

```
C$OMP BARRIER
```

The BARRIER directive has no clauses.

2.1.1.10 The ATOMIC directive

The ATOMIC directive ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The format of this directive is as follows:

```
C$OMP ATOMIC
```

The ATOMIC directive has no clauses.

2.1.1.11 The FLUSH

The FLUSH directive identifies synchronization points at which the implementation is required to provide a consistent view of memory. Thread-visible variables are written back to memory at the point at which this directive appears. The format of this directive is as follows:

```
C$OMP FLUSH [ ( list ) ]
```

The FLUSH directive has no clauses.

2.1.1.12 The ORDERED directive

The code enclosed within ORDERED and END ORDERED directives is executed in the order in which it would be executed in a sequential execution of the loop. These directives have the following format:

```
C$OMP ORDERED
```

```
block
C$OMP END ORDERED
```

The ORDERED directive has no clauses.

2.1.1.13 The THREADPRIVATE directive

The THREADPRIVATE directive makes named common blocks private to a thread but global within the thread. The format of this directive is as follows:

```
C$OMP THREADPRIVATE ( /cb/ [,/cb/]... )
```

The THREADPRIVATE directive has no clauses.

2.1.2 OpenMP Clauses

Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct. Not all of the following clauses are allowed on all directives, but the clauses that are valid on a particular directive are included with the description of the directive. The following sections describe the data scope attribute clauses.

2.1.2.1 The PRIVATE clause

The PRIVATE clause declares the variables in *list* to be private to each thread in a team. This clause has the following format:

```
PRIVATE ( list )
```

2.1.2.2 The SHARED clause

The SHARED clause makes variables that appear in the *list* shared among all the threads in a team. All threads within a team access the same storage area for SHARED data. This clause has the following format:

```
SHARED ( list )
```

2.1.2.3 The DEFAULT clause

The DEFAULT clause allows the user to specify a PRIVATE, SHARED, or NONE scope attribute for all variables in the lexical extent of any parallel region. This clause has the following format:

```
DEFAULT ( PRIVATE | SHARED | NONE )
```

2.1.2.4 The FIRSTPRIVATE clause

The FIRSTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in the *list* are subject to PRIVATE clause semantics described in section 2.1.2.1. In addition, private copies of the variables are initialized from the original object existing before the construct. This clause has the following format:

```
FIRSTPRIVATE ( list )
```

2.1.2.5 The LASTPRIVATE clause

The FIRSTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in the *list* are subject to PRIVATE clause semantics described in section 2.1.2.1. In addition, the thread that executes the sequentially last iteration of a loop updates the version of the object it had before the construct. This clause has the following format:

```
LASTPRIVATE ( list )
```

2.1.2.6 The REDUCTION clause

This clause performs a reduction on the variables that appear in *list*, with an operator or an intrinsic. The operator is one of the following: +, *, -, .AND., .OR., .EQV., or .NEQV., and intrinsic is one of the following: MAX, MIN, IAND, IOR, or IEOR. This clause has the following format:

```
REDUCTION ( { operator | intrinsic } : list )
```

2.1.2.7 The SCHEDULE clause

The SCHEDULE clause specifies how iterations of the DO loop are divided among the threads of the team. This clause has the following format:

```
SCHEDULE ( type [ , chunk ] )
```

Within the SCHEDULE syntax, *type* can be one of the following: STATIC, DYNAMIC, GUIDED, and RUNTIME.

When STATIC type is specified, iterations of the loop are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. If the chunk size is not specified, the iterations are divided equally in contiguous chunks among the threads.

When DYNAMIC type is specified, iterations of the loop are broken into pieces of a size specified by *chunk* as is the case with STATIC scheduling. However, the assignment of chunks to threads is done at run-time. As each thread finishes a chunk of the iteration space, it dynamically obtains the next set of iterations.

When GUIDED type is specified, the iterations of the loop are divided into chunks and assigned to threads at run-time as is the case with DYNAMIC scheduling. However, the size of the chunks is not constant. The chunk size is reduced exponentially as the chunks are assigned to threads. The value of *chunk* specifies the minimum number of iterations to dispatch each time, except when there are less than *chunk* number of iterations, at which point the rest are dispatched.

When RUNTIME type is specified, the specification of the scheme of scheduling is deferred until run time, and is determined by examining the OMP SCHEDULE environment variable.

2.1.2.8 The COPYIN clause

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 is to be copied to the thread private copies of the common block at the beginning of the parallel region. This clause has the following format:

```
COPYIN ( list )
```

2.1.2.9 The IF clause

When the IF clause is present in a parallel region, the enclosed code region is executed in parallel only if the *scalar_logical_expression* evaluates to .TRUE.. Otherwise, the parallel region is serialized. This clause has the following format:

```
IF ( scalar_logical_expression )
```

2.1.2.10 The ORDERED clause

The ORDERED clause can only be present in the DO directive when ordered sections are contained in the dynamic extent of the DO directive. For more information on ordered sections, see the ORDERED directive in Section 2.1.1.12. This clause has the following format:

```
ORDERED
```

2.1.3 OpenMP Environment Routines

The following is a list of OpenMP environment routines with a brief description of their functionality.

```
SUBROUTINE OMP_SET_NUM_THREADS( scalar_integer_expression )
```

The OMP_SET_NUM_THREADS subroutine sets the number of threads to use for the next parallel region.

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

The OMP_GET_NUM_THREADS function returns the number of threads currently in the team executing the parallel region from which it is called.

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

The OMP_GET_MAX_THREADS function returns the maximum value that can be returned by calls to the OMP_GET_NUM_THREADS() function.

```
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

The OMP_GET_THREAD_NUM function returns the thread number, within the team, that lies between 0 and OMP_GET_NUM_THREADS()-1, inclusive.

```
INTEGER FUNCTION OMP_GET_NUM_PROCS()
```

The OMP_GET_NUM_PROCS function returns the number of processors that are available to the program.

```
LOGICAL FUNCTION OMP_IN_PARALLEL()
```

The OMP_IN_PARALLEL function returns .TRUE. if it is called from the dynamic extent of a region executing in parallel, and .FALSE. otherwise.

```
SUBROUTINE OMP_SET_DYNAMIC ( scalar_logical_expression )
```

The OMP_SET_DYNAMIC subroutine enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
```

The OMP_GET_DYNAMIC function returns .TRUE. if dynamic thread adjustment is enabled and returns .FALSE. otherwise.

```
SUBROUTINE OMP_SET_NESTED ( scalar_logical_expression )
```

The OMP_SET_NESTED subroutine enables or disables nested parallelism.

```
LOGICAL FUNCTION OMP_GET_NESTED
```

The OMP_GET_NESTED function returns .TRUE. if nested parallelism is enabled and .FALSE. otherwise.

```
SUBROUTINE OMP_INIT_LOCK ( var )
```

The OMP_INIT_LOCAK subroutine initializes a lock associated with lock variable *var* for use in subsequent calls.

```
SUBROUTINE OMP_DESTROY_LOCK ( var )
```

The OMP_DESTROY_LOCK subroutine disassociates the given lock variable *var* from any locks.

```
SUBROUTINE OMP_SET_LOCK ( var )
```

The OMP_SET_LOCK subroutine forces the executing thread to wait until the specified lock is available.

```
SUBROUTINE OMP_UNSET_LOCK ( var )
```

The OMP_UNSET_LOCK subroutine releases the executing thread from ownership of the lock.

```
LOGICAL FUNCTION OMP_TEST_LOCK ( var )
```

The OMP_TEST_LOCK function tries to set the lock associated with the lock variable *var*. It returns .TRUE. if the lock was set successfully, and .FALSE. otherwise.

2.1.4 OpenMP Environment Variables

OpenMP environment variables control the execution of parallel code. They are briefly described bellow.

OMP_SCHEDULE

This variable applies only to DO and PARALLEL directives that have the schedule type RUNTIME. The scheduling type can be set at run time by setting this variable to any of the recognized schedule types.

OMP_NUM_THREADS

The OMP_NUM_THREADS environment variable sets the number of threads to use during execution, unless that number is explicitly changed by calling the OMP_SET_NUM_THREADS() subroutine.

OMP_DYNAMIC

The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

OMP_NESTED

The OMP_NESTED environment variable enables or disables nested parallelism.

2.2 The POSIX standard

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the UNIX operating system. The need for standardization arose because enterprises using computers wanted to be able to develop programs that could be moved among different manufacturer's computer systems without having to be recoded. UNIX was selected as the basis for a standard system interface partly because it was "manufacturer-neutral." However, several major versions of UNIX existed so there was a need to develop a common denominator system.

Informally, each standard in the POSIX set is defined by a decimal following the POSIX. Thus, POSIX.1 is the standard for an application program interface in the C language. POSIX.2 is the standard SHELL and utilities interface (that is to say, the user's command interface with the operating system). These are the main two interfaces, but additional interfaces, such as POSIX.4 for thread management, have been developed or are being developed. The POSIX interfaces were developed under the auspices of the IEEE.

POSIX.4 interface and a subset of its implementation, namely POSIX Standard 1003.1c-1995, are utilized extensively in this thesis. The standard covers the thread management routines, which are easily identified by their prefix *pthread*. These routines are used throughout the run-time library of the Open MP source-to-source compiler, as a mechanism for creation and

synchronization of threads, enforcement of mutual exclusion, and synchronization of execution using condition variables. All of the routines used by the run-time library are listed below:

Type	Description
pthread_t	Thread ID
pthread_mutex_t	Mutual Exclusion Lock (mutex)
pthread_cond_t	Condition variable (cv)
pthread_attr_t	Thread attribute

Table 1 POSIX.1c types

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*entry)(void *),
                    void *arg );
    Function: Create a new thread of execution.
    Errors: EAGAIN, EINVAL
    Note: Maximum number of PTHREAD_THREADS_MAX threads per process.


int pthread_join( pthread_t thread,
                  void **status );
    Function: Synchronize with the termination of a thread.
    Errors: EINVAL, ESRCH, EDEADLK
    Note: This function is a cancellation point.


pthread_t  pthread_self( void );
    Function: Return the thread ID of the calling thread.
    Errors: none
int pthread_attr_init( pthread_attr_t *attr );
    Function: Initialize a thread attribute object.
    Errors: ENOMEM


int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr );
    Function: Initialize a mutex.
    Errors: EAGAIN, ENOMEM, EPERM, EBUSY, EINVAL


int pthread_mutex_destroy( pthread_mutex_t *mutex );
    Function: Destroy a mutex.
    errors: EBUSY, EINVAL
```

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
    Function: Acquire the indicated mutex.
    Errors: EINVAL, EDEADLK


int pthread_mutex_trylock( pthread_mutex_t *mutex );
    Function: Attempt to acquire the indicated mutex.
    Errors: EINVAL, EBUSY, EINVAL


int pthread_mutex_unlock( pthread_mutex_t *mutex );
    Function: Release the (previously acquired) mutex.
    Errors: EINVAL, EPERM


int pthread_cond_init( pthread_cond_t *cond,
                       const pthread_condattr_t *attr );
      Function: Initialize a condition variable.
    Errors: EAGAIN, ENOMEM, EBUSY, EINVAL


int pthread_cond_destroy( pthread_cond_t *cond );
    Function: Destroy a condition variable.
    Errors: EBUSY, EINVAL


int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
    Function: Block on the specified condition variable.
    Errors: EINVAL
    Note: This function is a cancellation point


int pthread_cond_broadcast( pthread_cond_t *cond );
    Function: Unblock all threads currently blocked on the specified
                condition variable.
    Errors: EINVAL
```

2.3 The POLARIS compiler

The implementation of the OpenMP compiler was done with the help of the POLARIS compiler from the University of Illinois at Urbana Champaign [3]. Although POLARIS includes facilities to detect parallelism in sequential programs and generate parallel code, these facilities were not used. Rather, the core of the POLARIS compiler was used to parse, represent, and manipulate FORTRAN code. This allowed us to add parsing of OpenMP directives, to restructure the input program by packaging parallel regions as subroutines, and to insert calls to the run-time system to create and schedule threads. The remainder of this section gives an overview of POLARIS, focusing on its components most relevant to our work.

The POLARIS compiler provides a basic compiler development infrastructure that includes a powerful internal representation, and general purpose utilities. It also provides implementations for many effective program transformations, including inline expansion, interprocedural constant propagation, induction variable substitution, propagation of constants, symbolic constants, dead code and dead store elimination, array privatization, reduction substitution, and data dependence analysis. Finally, it provides a postpass to select loops for parallelization and output the transformed program in the FORTRAN dialects (including directives) of such systems as the SGI Challenge and Convex Exemplar. Figure 4 depicts POLARIS's major functional units.

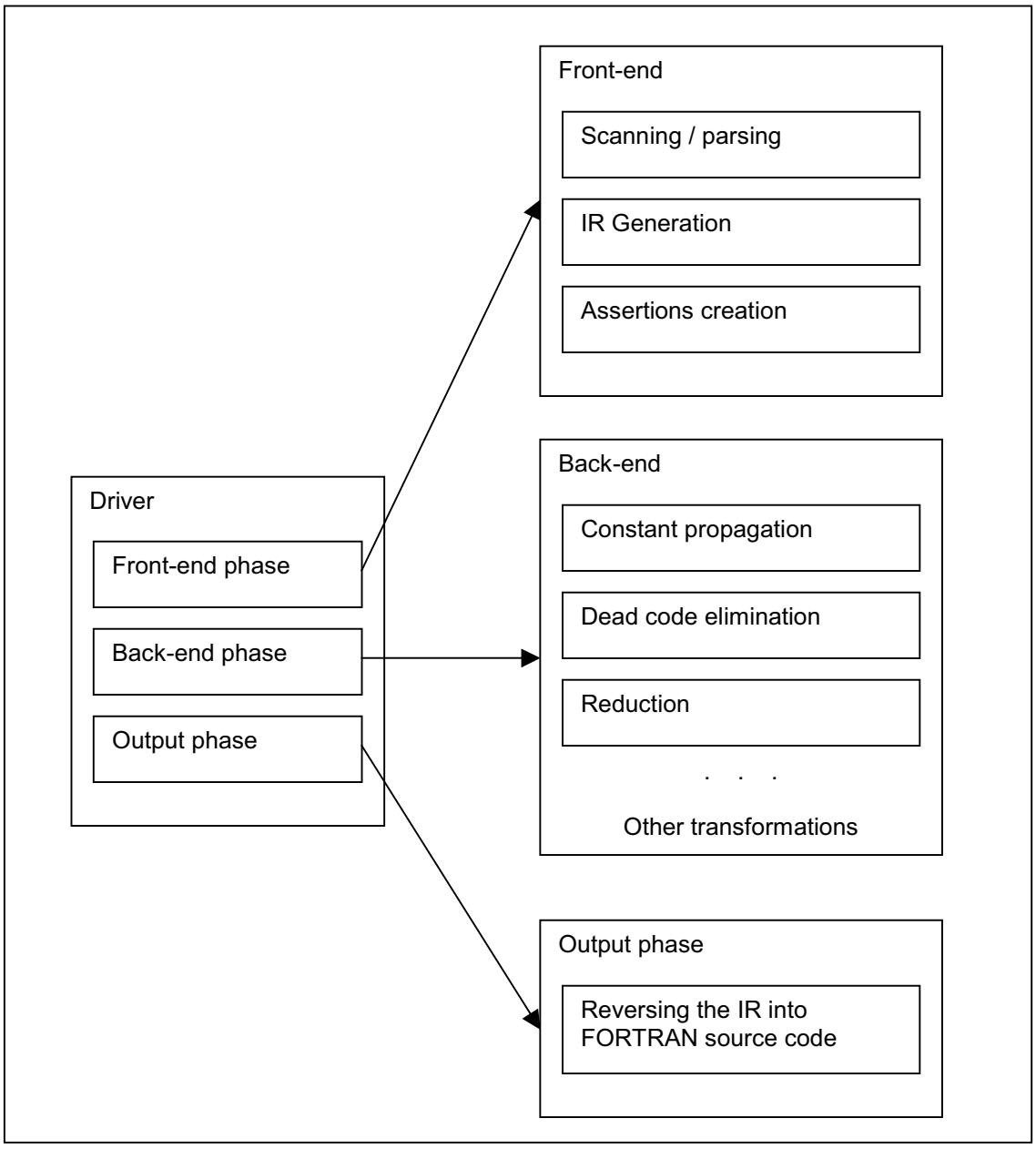

Figure 4 POLARIS architecture.

POLARIS is implemented in C++ which allows clear structure and high data abstraction through the representation of most constructs with objects of classes which derive from broad categorical class hierarchies. For example, a FORTRAN DO statement is represented as an object of class DoStmt, which is derived from class Statement, the base of the statement class hierarchy. Most individual services for general class hierarchies (e.g., the statement and expression class hierarchies) are implemented using virtual member functions, which allows pointers and references to such constructs to be left in the most general form.

The POLARIS collection class hierarchy provides a family of template classes, which implement lists, sets, trees, and maps of class objects, and provides the concept of object ownership, iterators and mutators, and reference counting to maintain the semantic correctness of a collection. POLARIS also contains many different general-purpose utilities, such as control flow and loop normalizers and expression search-and-replace routines.

POLARIS represents FORTRAN programs as Program objects containing one or more ProgramUnit objects, each of which represents an individual FORTRAN routine. Each ProgramUnit object contains a number of bookkeeping objects (e.g., a symbol table, DATA statements, FORMAT statements), and a StmtList object to represent the executable statements of a routine and their structure. A StmtList object owns various class objects derived from class Statement, the base of the statement class hierarchy, which represent the statements of the routine.

A statement contains information about adjoining statements and sets representing references to variables, and owns a list of “top level” expressions, which represent the top of an abstract syntax tree. An expression is represented by an object of a class derived from class Expression, the base of the expression class hierarchy, and implements all or part of an abstract syntax tree representing an arbitrarily complex expression.

POLARIS’s front end also has a built in capability to parse FORTRAN comments or comment like language directives. These directives must have their first lexical token starting with letter "C". This feature provides simple and effective means of adding custom directives into the language. The compilers that understand these directives will act appropriately when the directives are encountered, while other FORTRAN compilers will simply ignore them as if they were comments.

POLARIS’s parsing mechanism is done with a help of a toolset called Purdue Compiler Construction Tool Set (PCCTS). PCCTS consists of two tools, ANTLR (Another Tool for

Language Recognition) and DLG (DFA-based Lexical analyzer Generator). ANTLR is a functional equivalent of YACC (Yet Another Compiler Compiler), and it generates parsers from a BNF-like grammar description. DLG is a functional equivalent of LEX (Lexical Analyzer), and it generates lexical analyzers (scanners) from a set of regular-expression rules.

The POLARIS uses the ANTLR parser for the purpose of parsing FORTRAN programs. The ANTLR allows for quick and simple language extensions through directive embedding. All directive specifications, directive parsing rules, and modification code, are contained in a file called *parser.g*. When ANTLR is invoked, it reads this file and creates C++ code that will identify, among other FORTRAN program constructs, custom directives according to the parsing rules specified. Upon recognizing a comment as a built-in directive, a *directive specific handler* routine is invoked. The handler stores the directive and other pertinent information into one or more *assertions*. An assertion is represented by an object of a class derived from class Assertion, the base of the assertion class hierarchy. Assertion objects describe the type of the directive, and hold additional information and parameters that are necessary to process the directives during the back-end pass. Once an Assertion object is created, the compiler generally adds it to a Statement object. Assertions can be attached to a Statement object at the "beginning", or "end" of the Statement. This abstract positioning is meant to help compiler writers store directive information in the lexical order the program was parsed. For example, if a directive lexically precedes a FORTRAN statement, an Assertion object representing this directive is created and attached at the "beginning" of the Statement object representing the FORTRAN statement.

Chapter 3

Design and Implementation

This chapter provides the implementation details of our source-to-source compiler. The first section describes the implementation of the front end of the compiler. The second section depicts various OpenMP constructs and the way in which they are handled by the back end. The third section describes the run-time library with an in-depth look at run-time routines that provide support for parallelism.

3.1 The front end pass

In order to be recognized as an OpenMP directive and not ordinary comment or other directive, the first token of an OpenMP directive is required to be one of the following sentinels: OMP, COMP, or *$OMP. When the parser finds a directive that begins with such a sentinel, it stores its contents into an object of AssertOMP type. AssertOMP is a container class created to hold the type of the directive encountered (e.g. PARALLEL, DO, MASTER, etc.), and the information about the directive's clauses. It is derived from the ExpressionAssertion class, which is provided by POLARIS. Each AssertOMP assertion is stored at the "beginning" of the POLARIS' Statement object that represents the FORTRAN statement lexically following the OpenMP directive.

In order to implement OpenMP directive parsing, POLARIS's parsing rules had to be extended as follows. First, a new token definition that represents all three acceptable OpenMP sentinels was added to the parsing language. This token is then used as part of the parsing rules that try to match it with the first token of every comment line. If a match is found, the parser discovered an OpenMP directive. Figure 5 shows the definition of the OMP_TOKEN token that was inserted into the parsing rule file.

```
#token OMP_TOKEN     "(\C|\!|\*)\$OMP"  << zzmode(NORMAL); >>
```

Figure 5 OpenMP sentinel definition.

The parsing rules responsible for the detection of various directives are contained within POLARIS's `parser_comment_or_directive` routine. This routine has been extended to incorporate the recognition of OpenMP directives. Since the body of the routine is essentially a "switch" statement that identifies different directives and then routes their parsing to different handler routines, a new "case" that checks for OpenMP directives was added to it. This case forwards the parsing of all comment lines whose first token is the `OMP_TOKEN` to an OpenMP handler routine. Figure 6 shows an excerpt from the `parser_comment_or_directive` routine illustrating the implementation of the routing of CSRD, CONVEX, and OpenMP directives to their respective parsing handlers. The figure shows that all lines of code that are recognized as OpenMP directives, are further parsed by the `parser_omp_directive` routine.

```
parser_comment_or_directive[ParserContext &context, const char *line]
"Fortran comment or directive"
   :  CSRD_TOKEN
          parser_csrd_directive[ context ]
          << zzmode( START ); >>
   |  CONVEX_TOKEN
          parser_convex_directive[ context ]
          << zzmode( START ); >>
  .  .  .
   |  OMP_TOKEN
          parser_omp_directive[ context, line ]
          << zzmode( START ); >>
  .  .  .
```

Figure 6 Polaris's parsing routine responsible for parsing comments and directives.

The `parser_omp_directive` routine is responsible for the identification of the OpenMP directive types (i.e. `PARALLEL`, `SECTIONS`, `SINGLE`, etc.), parsing of the clauses used in conjunction with the directive, and creation of AssertOMP assertions that hold the information about the directive and its clauses. Figure 7 shows a piece of code extracted from the `parser_omp_directive` routine that illustrates how the routine handles the PARALLEL directive.

```
parser_omp_directive[ParserContext &context, const char *line ]
"OMP directive"
   :  <<
         ExpressionList *list = 0;
         Statement *s = &context.s_next;
         AssertOMP *a = new AssertOMP;
      >>
      (
        "PARALLEL"
        (
            ( "DO"
                parser_omp_parallel_do_clause_list[context] > [list]
                <<
                   a->set_directive(parallel_do_dir);
                   s->assertions().ins_last( a );
                   list_absorb(a->arg_list_guarded(), list);
                >>
            |
              "SECTIONS"
                parser_omp_parallel_sections_clause_list[context] > [list]
                <<
                   a->set_directive(parallel_sections_dir);
                   s->assertions().ins_last( a );
                   list_absorb(a->arg_list_guarded(), list);
                >>
            )
            |
            parser_omp_parallel_clause_list[context] > [list]
            <<
               a->set_directive(parallel_dir);
               s->assertions().ins_last( a );
               list_absorb(a->arg_list_guarded(), list);
            >>
        )
    .   .   .
```

Figure 7 OpenMP parsing handler routine.

The code consists of two parts: an initialization component that creates assertion objects and other variables, and the parsing rule component that tests for the existence of the `PARALLEL` token. When the `PARALLEL` token is found, the parser looks for the `DO` or `SECTIONS` tokens that are allowed to be specified in conjunction with the `PARALLEL` token. Depending on the type of directive discovered, an appropriate parsing handler is then invoked. The handler collects information about the given directive's clauses and returns it as an `ExpressionList` object. This object is then finally stored as part of the `AssertOMP` assertion.

Most OpenMP directives allow for optional clauses that specify the functionality of the directives in more detail. When a parser encounters an optional clause, it parses out its contents and stores them into AssertOMP's argument list as OMPClauseExpr objects. OMPClauseExpr class provides the ability to store the type of the clause (e.g. PRIVATE, SHARED,

FIRSTPRIVATE, etc.), and variables specified in the clause. It is a derived class of the Expression class provided by POLARIS to represent expressions. If the directive in the previous example were a `PARALLEL DO` directive, the `parser_omp_directive` routine would first identify the `PARALLEL` and `DO` tokens, and then invoke the `parser_omp_parallel_do_clause_list` clause parsing routine. This routine parses the clause list of the `PARALLEL DO` directive collecting information about the specified clauses. Figure 8 provides a shortened version of the routine that illustrates how the routine handles `PRIVATE, SHARED,` and `DEFAULT` clauses.

```
parser_omp_parallel_do_clause_list[ParserContext &context]
            > [ExpressionList *list]
"OMP PARALLEL DO Clause list"
   :
      <<
         $list = new ExpressionList;
         Expression *if_exp;

         // Initialize the list of clauses
         OMPClauseExpr *pr_cl=new OMPClauseExpr(private_clause);
         OMPClauseExpr *sh_cl=new OMPClauseExpr(shared_clause);
         OMPClauseExpr *de_cl=new OMPClauseExpr(default_clause);

... other clause initialization

      >>

      (
         "PRIVATE"
         "\("
         parser_omp_collect_symbols[context, pr_cl->arg_list()]
         "\)"
      | "SHARED"
         "\("
         parser_omp_collect_symbols[context, sh_cl->arg_list()]
         "\)"
      | "DEFAULT"
         parser_omp_default_clause[context, de_cl->arg_list()]

... other parsing rules

      ) *

      <<
         $list->ins_last(pr_cl);
         $list->ins_last(sh_cl);
         $list->ins_last(de_cl);

... other OMPClauseExpr expression insertion into list expression

      >>
   ;
```

Figure 8 PARALLEL DO clause parsing routine.

The top part of the routine shows the initialization section that is responsible for the creation of the OMPClauseExpr objects used to store information about each of the clauses. The code that follows the initialization section is the parsing rule section, which identifies the specified clauses, and invokes appropriate routines to parse out relevant information about each clause. In the case of the `PRIVATE` and `SHARED` clauses, POLARIS's `parser_omp_collect_symbols` routine is invoked to collect the symbols that are specified as private or shared. The DEFAULT clause on the other hand, is handled by the `parser_omp_default_clause` routine shown in Figure 9. This routine is used to record the type of the scope attribute (`PRIVATE`, `SHARED`, or `NONE`) specified for this clause.

```
parser_omp_default_clause[ParserContext &context, List<Expression> &list]
"OMP DEFAULT clause"
   :
      <<
         Expression *e_id = 0;
         default_clause_value def_val;
      >>

      "\("
      (
        "PRIVATE"
        << def_val = default_clause_private; >>
      | "SHARED"
        << def_val = default_clause_shared; >>
      | "NONE"
        << def_val = default_clause_none; >>
      )
      "\)"
      <<
         if (list.entries() != 0)
             p_abort("DEFAULT clause can only be used once.");
         e_id = new IntConstExpr(def_val);
         list.ins_last(e_id);
      >>
   ;
```

Figure 9 DEFAULT clause parsing routine.

The front end does not store any information about the environment routines, or environment variables. All calls to environment routines are left unchanged, and are explicitly handled at run-time by the OpenMP run-time library. The library provides all environment routine implementations, and is responsible for handling of environment variables.

3.2 The back end pass

This section presents the parallelization process, and the principles used by the OpenMP compiler for the translation of the OpenMP directives into FORTRAN code with embedded OpenMP run-time library calls. The section is further divided into two subsections: the first one that describes in detail the handling of the OpenMP directives and their clauses, and the second one that depicts the OpenMP run-time library implementation and its interface. Examples are provided throughout this section to further clarify specific implementation details.

3.2.1 Directive Implementation

The OpenMP back end phase is designed as a one-pass pass over the intermediate representation (IR) of the program generated by the front end. It is invoked by the driver function immediately after the front end finishes its task. Its function is to process each program unit by transforming the code marked by the OpenMP assertions. This is accomplished by first, searching for the existence of the OpenMP assertions in each Statement object, and then invoking an appropriate handler method on the statements effected by the given assertion. The handler's functionality is to modify the IR by changing, removing, and inserting statements and variables according to the needs of the OpenMP assertion. The following sections describe how each OpenMP directive is processed via their handler methods.

3.2.1.1 The PARALLEL directive

A parallel region, by definition, is a block of code that is to be executed by multiple threads in parallel. In order to implement this functionality, this block of code first needs to be extracted from the original procedure, and then repackaged as a new stand-alone subroutine. In its place, a call is made to the F_PTHREAD_CREATE_THREADS run-time routine that takes the address of the new subroutine as an argument. This run-time routine then creates of a team threads that will execute the new subroutine, and synchronizes the threads after they have completed their tasks (see Section 3.2.2.1 for more information about the run-time thread creation routines). The PARALEL directive code transformation is illustrated in Figure 10 for a sample program.

All variables designated as private in the PRIVATE clause of the PARALLEL directive are handled by simply declaring them 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

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

```
      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
```

Figure 10 Transformation of a FORTRAN program that uses a PARALLEL directive.

across different instances of the subroutine. Variables designated in a FIRSTPRIVATE clause are declared in the new subroutine and initialized to the values of the original variables. The initialization values are obtained via common block temporary variables that take on the values of the original variables. Figure 11 shows an example of the code transformations that involves handling of PRIVATE, SHARED, and FIRSTPRIVATE variables.

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

```
      SUBROUTINE omp_sub1(omp_thr_num)
      INTEGER*4 i, j, k, 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
```

Figure 11 Transformation of a FORTRAN program that uses a PARALLEL directive and PRIVATE, SHARED, and FIRSTPRIVATE clauses.

Although shared variables can be generally made available to the new subroutine via common blocks, FORTRAN 77 language does not permit function and subroutine parameters to reside in common blocks. If a function or subroutine parameter is declared to be shared, the

source-to-source compiler passes this variable to the new subroutine as a parameter of the F_PTHREAD_CREATE_THREADS run-time routine. This routine then creates the parallel threads that subsequently execute the new subroutine, passing it the shared variables as parameters. The routine takes the following arguments: the address of the new subroutine, the number of shared variables passed as parameters, and the shared variables. Example shown in Figure 12 illustrates this case.

```
      program                    SUBROUTINE omp_sub1(omp_thr_num, i)
      integer i                  INTEGER*4 i, j, omp_thr_num
      i = 5                      COMMON /omp_cb1/ j
      call test(i)               j = j+i
      end                        END

      subroutine test(i)         PROGRAM
      integer i, j               EXTERNAL test
      j = 1                      INTEGER*4 i
c$OMP PARALLEL                   i = 5
      j = j + i                  CALL test(i)
c$OMP END PARALLEL               STOP
      print *, i, j              END
      end
                                 SUBROUTINE test(i)
                                 EXTERNAL f_pthread_create_threads
                                 EXTERNAL omp_sub1
                                 INTEGER*4 i, j
                                 COMMON /omp_cb1/ j
                                 j = 1
                                 CALL f_pthread_create_threads(omp_sub1,1,i)
                                 PRINT *, i, j
                                 RETURN
                                 END
```

Figure 12 Original and transformed programs where the shared variable i is passed as a parameter, and the shared variable j is shared through a common block.

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. When the expression is true, multiple thread call is invoked, and when false, one thread call is invoked. Figure 13 shows an example of a parallel region that contains an IF clause.

The implementation of the REDUCTION clause is slightly more complicated than the implementation of other clauses. All variables that appear in a REDUCTION clause must be shared in the enclosing context. Therefore, 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.

```
      program test
      logical l,k
      l = .FALSE.
      k = .TRUE.
      print *, "Started"
c$OMP PARALLEL IF(l .OR. k)
      print *, "Hello world"
c$OMP END PARALLEL
      print *, "Finished"
      end
```

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

      PROGRAM test
      EXTERNAL f_pthread_create_one_thread,
      EXTERNAL f_pthread_create_threads
      EXTERNAL omp_sub1
      LOGICAL*4 k, l
      l = .FALSE.
      k = .TRUE.
      PRINT *, 'Started'
      IF (k.OR.l) THEN
       CALL f_pthread_create_threads(
                              omp_sub1, 0)
      ELSE
       CALL f_pthread_create_one_thread(
                              omp_sub1, 0)
      ENDIF
      PRINT *, 'Finished'
      STOP
      END
```

Figure 13 Transformation of a FORTRAN program that uses a PARALLEL directive and an IF clause.

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

```
      SUBROUTINE omp_sub1(omp_thr_num)
      EXTERNAL omp_set_lock
      EXTERNAL omp_unset_lock
      INTEGER*4 i, omp_mutex1, omp_reduce2
      INTEGER*4 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
```

Figure 14 Transformation of a FORTRAN program that uses a PARALLEL directive and a REDUCTION clause.

Another variable is then used to represent the original variable that is accessible through a common block. This variable is updated at the end of the parallel region to reflect the use of the specified reduction operator. An example of the implementation of PARALLEL with a REDUCTION clause is shown in Figure 14.

When nested parallelism is enabled via the OMP_NESTED environment variable, or the OMP_SET_NESTED subroutine, the OpenMP standard allows the nested parallel regions to deploy additional threads to form a team. The number of threads used to execute nested parallel regions is implementation dependant. As a result, OpenMP compliant implementations are allowed to serialize nested parallel regions when nested parallelism is enabled. This is the approach taken by our source-to-source compiler.

3.2.1.2 The DO directive

The DO directive specifies that the iterations of the immediately following DO loop are to be executed by parallel threads. The iterations of the loop are distributed across threads that already exist. This directive can be used in conjunction with six optional clauses: SCHEDULE, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, and ORDERED.

The SCHEDULE clause specifies how iterations of the DO loop are to be divided among the threads of the team. 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. Figures 16 and 17 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, which is necessary because there might be multiple chunks of iterations that each thread will work on.

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

Figure 15 Initial loop.

```
num_of_threads = OMP_GET_NUM_THREADS()
thread_num = OMP_GET_THREAD_NUM()
DO ii=0, (limit-init)/(chunk*stemp*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
```

Figure 16 Result after static transformations have been applied and chunk size was specified.

```
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
```

Figure 17 Result after static transformations have been applied and chunk size was not specified.

- DYNAMIC. An example of the implementation of this scheme is shown in Figure 18. 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 should execute.
- GUIDED. The implementation of this scheme is similar to that of DYNAMIC. The only difference is that the chunk size is computed using the following rule: Iteration size = number of iterations left / number of threads
- RUNTIME. The implementation of this scheme is also similar to that of DYNAMIC. However, the actual scheme of scheduling is set when the initialization 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.

```
    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
```

Figure 18 Result after dynamic transformations have been applied.

The handling of the PRIVATE, FIRSTPRIVATE, and REDUCTION clauses of the DO directive is done in a similar matter as with the PARALLEL directive: variables that appear in the PRIVATE clause are renamed and redeclared in the procedure containing the directive, FIRSTPRIVATE variables are treated as PRIVATE variables that are initialized above the DO loop, and REDUCTION variables are updated at the end of the parallel region using the specified reduction operator.

Variables that appear in the LASTPRIVATE clause are also treated as PRIVATE variables. In addition, the thread that executes the sequentially last iteration of the DO loop needs to update the version of the LASTPRIVATE variable it had before the DO construct. This is implemented by inserting an IF statement after the DO loop. This statement guards the LASTPRIVATE variable update, and only allows the thread that executes last iteration to do the updating. Figures 19 and 20 illustrate a full DO directive code transformation.

```
      program test
      integer*8 i,j
      print *, "Started"
      j = 0
c$OMP PARALLEL
c$OMP DO SCHEDULE(DYNAMIC,100) PRIVATE(i) REDUCTION(+:j) LASTPRIVATE(i)
      do i=1, 10000000, 1
         j = j + 1
      end do
c$OMP END PARALLEL
      print *, i, j
      print *, "Finished"
      end
```

Figure 19 Original FORTRAN program with an OMP DO directive and a variety of clauses.

```
      SUBROUTINE omp_sub1(omp_thr_num)
      EXTERNAL f_omp_barrier, f_omp_get_sched_vars, f_omp_init_sched
      EXTERNAL f_pthread_mutex_runtime_lock
      EXTERNAL f_pthread_mutex_runtime_unlock, omp_get_thread_num
      INTEGER*4 f_omp_get_sched_vars, omp_do_init6, omp_do_limit7
      INTEGER*4 omp_do_step8, omp_get_thread_num, omp_thr_num
      INTEGER*4 omp_thr_num5
      INTEGER*8 i, j, omp_do_prv1, omp_do_prv2, omp_do_prv4
      COMMON /omp_cb2/ i
      COMMON /omp_cb1/ j
      omp_do_prv4 = 0
      omp_thr_num5 = omp_get_thread_num()
      CALL f_omp_init_sched(1, 1, 10000000, 1, 100, omp_thr_num5)
      omp_do_init6 = 1
      omp_do_limit7 = 10000000
      omp_do_step8 = 1
    1 CONTINUE
      IF (f_omp_get_sched_vars(omp_do_init6, omp_do_limit7, omp_thr_num5
     *).EQ.0) THEN
       GOTO 2
      ENDIF
      DO omp_do_prv1 = omp_do_init6, omp_do_limit7, omp_do_step8
       omp_do_prv4 = omp_do_prv4+1
      ENDDO
      GOTO 1
    2 CONTINUE
      CALL f_pthread_mutex_runtime_lock()
      j = j+omp_do_prv4
      CALL f_pthread_mutex_runtime_unlock()
      omp_temp3 = 10000000
      IF (omp_do_prv1.EQ.1+omp_temp3*1) THEN
       i = omp_do_prv2
      ENDIF
      CALL f_omp_barrier()
      END

      PROGRAM test
      EXTERNAL f_pthread_create_threads, omp_sub1
      INTEGER*8 i, j
      COMMON /omp_cb2/ i
      COMMON /omp_cb1/ j
      PRINT *, 'Started'
      j = 0

      CALL f_pthread_create_threads(omp_sub1, 0)
      PRINT *, i, j
      PRINT *, 'Finished'
      STOP
      END
```

Figure 20 Results after all DO transformations have been applied.

The DO directive also specifies that an implicit synchronization barrier has to be placed at the end of the scheduled loop. Figure 20 shows that after the transformations have been performed, the end of the newly created subroutine contains a call to the F_OMP_BARRIER run-time routine, which implements the required barrier synchronization.

3.2.1.3 The SECTIONS directive

The SECTIONS directive is a non-iterative work-sharing construct that specifies the enclosed SECTIONS of the code are to be executed by parallel threads. To ensure each SECTION in executed by exactly one thread, an if statement is inserted around every SECTION in the SECTIONS 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.

The optional clauses that can be used in conjunction with this directive (PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION) are handled in a manner similar to how they would be handled in a PARALLEL or DO directive case. Figures 21, and 22 show the code transformations that occur when a SECTIONS directive is used.

```
      program test
      integer*8 i, j
      print *, "Started"
      i = 1
      j = 2
c$OMP PARALLEL
c$OMP SECTIONS
c$OMP SECTION
      i = i + 1
c$OMP SECTION
      j = j + 1
c$OMP END SECTIONS
c$OMP END PARALLEL
      print *, "Finished: ", i, j
      end
```

Figure 21 Original FORTRAN program with a SECTIONS directive.

```
      SUBROUTINE omp_sub1(omp_thr_num)
      INTRINSIC mod
      EXTERNAL f_omp_barrier, omp_get_num_threads, omp_get_thread_num
      INTEGER*4 omp_get_num_threads, omp_get_thread_num
      INTEGER*4 omp_no_of_thr1, omp_thr_num, omp_thr_num2
      INTEGER*8 i, j
      COMMON /omp_cb2/ i
      COMMON /omp_cb1/ j
      omp_no_of_thr1 = omp_get_num_threads()
      omp_thr_num2 = omp_get_thread_num()
      IF (MOD(1, omp_no_of_thr1).EQ.omp_thr_num2) THEN
       i = i+1
      ENDIF
      IF (MOD(2, omp_no_of_thr1).EQ.omp_thr_num2) THEN
       j = j+1
      ENDIF
      CALL f_omp_barrier()
      END
```

```
      PROGRAM test
      EXTERNAL f_pthread_create_threads, omp_sub1
      INTEGER*8 i, j
      COMMON /omp_cb2/ i
      COMMON /omp_cb1/ j
      PRINT *, 'Started'
      i = 1
      j = 2
      CALL f_pthread_create_threads(omp_sub1, 0)
      PRINT *, 'Finished: ', i, j
      STOP
      END
```

Figure 22 Resulting code after SECTIONS transformations have been applied.

3.2.1.4 The SINGLE directive

The SINGLE directive specifies that the enclosed code is to be executed by only one thread of the team. It is handled exactly as the SECTIONS directive except that the thread that is allowed to execute this block is the master thread. All of the optional directive clauses (PRIVATE, and FIRSTPRIVATE) are handled in a manner similar to how they would be handled in a PARALLEL or DO directive case.

3.2.1.5 The PARALLEL DO directive

The PARALLEL DO directive provides a shortcut form for specifying a PARALLEL region that contains a single DO directive. The back-end implements this directive by dividing it into two new constructs: a PARALLEL directive followed by a DO directive. The clauses of the PARALLEL DO directive are separated, and assigned to two new constructs as follows. The PARALLEL directive is assigned: SHARED, DEFAULT, IF, and COPYIN clauses. The DO directive gets: PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, SCHEDULE, and ORDERED clauses. All FIRSTPRIVATE, LASTPRIVATE, and REDUCTION variables are also copied into the SHARED list of the PARALLEL directive. The new constructs are then handled separately as described in the PARALLEL, and DO sections.

3.2.1.6 The PARALLEL SECTIONS directive

The PARALLEL SECTIONS directive provides a shortcut form for specifying a parallel region that contains a single sections directive. Its handling is done similarly to PARALLEL DO directive: it is divided into a PARALLEL directive followed by a SECTIONS directive. Its

clauses are separated and assigned to two new constructs. The PARALLEL directive is assigned: SHARED, DEFAULT, IF, and COPYIN clauses. The SECTIONS directive gets: PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses. In addition, all FIRSTPRIVATE, LASTPRIVATE, and REDUCTION variables are copied into the shared list of the PARALLEL directive. The new constructs are then handled separately as described in the PARALLEL, and SECTIONS sections.

3.2.1.7 The MASTER directive

The code enclosed within the MASTER directive block is executed by the master thread of the team. It is handled in the same manner as the SINGLE directive.

3.2.1.8 The CRITICAL directive

The CRITICAL directive restricts access to the enclosed code to only one thread at a time. This is accomplished by calling F_OMP_SET_CRITICAL_LOCK, and F_OMP_UNSET_CRITICAL_LOCK runtime routines that set and unset a lock, so only one thread can be executing enclosed code at the same time.

3.2.1.9 The BARRIER directive

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. This is accomplished by calling F_OMP_BARRIER run-time procedure that puts the threads on a conditional queue until all threads have called this routine.

3.2.1.10 The ATOMIC directive

The ATOMIC directive ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. It is handled in the same matter as the CRITICAL directive.

3.2.1.11 The FLUSH directive

Due to the fact that the FLUSH directive identifies a synchronization point at which the running program must provide a consistent view of memory, the back end is not able to provide a

cross platform implementation of the directive. Instead, the user is provided with the information about the directive in a form of an assertion, and a hook in the code where a platform specific implementation needs to be inserted.

3.2.1.12 The ORDERED directive

The back end currently does not implement the ORDERED directive. This is a seldom used directive as its definition inhibits true parallelism (the directive provides serialization of code across iterations of a parallel loop). Since the directive has not been encountered in any of the benchmarks, and given the limited time for the implementation of this project, the directive is only implemented by the front end. At this time, the back end recognizes the ordered assertion as created by the front end, but it does not implement its functionality.

3.2.2 The Run-time Library

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

The run-time routines are divided into four different groups according to their functionality: thread creation routines, thread synchronization routines, scheduling routines, and OpenMP environment routines. The following sections describe each of the groups and provide implementation details of some of the more important routines within each group.

3.2.2.1 The thread creation routines

The thread creation routines are responsible for creating, initializing, running, and joining a team of threads. Figure 23 shows the thread creation routines exported by the run-time library.

```
void f_pthread_create_threads(void *(*f)(void *), long *argcount, ...);
void f_pthread_create_one_thread(void *(*f)(void*), long *argcount, ...);
```

Figure 23 The prototypes of the thread creation routines.

The calls to the *f_pthread_create_threads* routine are created by the source-to-source compiler phase, and placed at the start of a parallel region. When the routine is executed, it initializes the run-time status information, stores pointers to the shared variables, and creates the threads that will run the routine specified as a parameter. Although the number of threads the routine creates is dependent on the value of the OMP_NUM_THREADS environment variable, a dynamic adjustment of this number can be achieved by calling the OMP_SET_NUM_THREADS routine prior to entering the parallel region.

As each thread is executed, it first runs the run-time library routine called *run_me*. This routine serves as a proxy between the *f_pthread_create_threads* routine and the routine that is required to be executed in parallel. This proxy routine is used as a work around the POSIX threads library restriction, that limits the number of parameters that can be passed to a routine that is executed as a new thread's start routine. A work around this issue is required due to the feature of the source-to-source compiler that occasionally passes shared variables to the parallel routine using the parameters passing method, instead of the common block method (for more details on this feature see section 3.2.1.1). In such a case, the *f_pthread_create_threads* routine stores the address of the parallel routine, and pointers to the shared variables passed as parameters, into the run-time storage. The routine then creates a team of threads that will run the *run_me* routine. The *run_me* routine retrieves the address of the parallel routine, and the pointers to the shared variables, and calls the parallel routine passing it the shared variables as parameters. An example of the execution path of a sample OpenMP program is shown in Figure 24.

The following sequence illustrates the execution path resulting from running the program shown in Figure 9 section 3.2.1.1.

```
      SUBROUTINE test(i)
      ...
      CALL f_pthread_create_threads(omp_sub1, 1, i)
      ...
      END
```

The subroutine *test* first calls the *f_pthread_create_threads* routine to create and initialize parallel threads. Two parameters are passed to the subroutine: *omp_sub1* - the subroutine that is to be executed in parallel, and a shared variable *i*.

```
void f_pthread_create_threads(void *(*f)(void *), long *argcount, ...) {
   /* Store the parallel routine and shared parameters into the
      run-time storage */
   g_f = f;
   for (i=0; i<*argcount; ++i)
      g_parameters[i] = va_arg(p, long);
   ...
```

```
   /* Set program state */
   ...
   /* Create threads */
   ...
   pthread_create(&threads[i],&attr,(void *)&run_me,(void *)&thr_num[i]);
   ...
}
```

When the run-time routine *f_pthread_create_threads* is called, it stores all shared variables into the run-time storage and creates threads that will execute the *run_me* routine.

```
static void run_me(void *thread_num) {
   ...
   (*g_f)(thread_num, g_parameters[0]);
   ...
}
```

When the run-time routine *run_me* is executed by the threads, it calls *omp_sub1* passing it its thread number, and the shared variable *i*.

```
      SUBROUTINE omp_sub1(omp_thr_num, i)
      ...
      END
```

Finally, the subroutine *omp_sub1* executes the parallel code.

Figure 24 Execution path of a sample program.

After the *f_pthread_create_threads* routine creates all of the threads, it blocks (using a *join* call), and waits for all threads to finish. Finally, it removes all information about the program state which effectively resets the run-time into the initial state

The *f_pthread_create_one_thread* routine, on the other hand, is only responsible for creating, initializing, and running of one thread. It is identical to the *f_pthread_create_threads* routine, except that it is only used in cases when a parallel region contains an IF clause. When this clause is false (determined dynamically), the region needs to be serialized, and therefore only one thread needs to be created.

3.2.2.2 The Synchronization Routines

The synchronization group of routines consists of the OpenMP lock routines, routines used to implement CRITICAL and BARRIER directives, and the routines used to provide thread synchronization for the DO, SECTIONS, and SINGLE directives. The routine prototypes are listed in Figures 25 and 26.

```
void omp_init_lock(int *mutex);
void omp_destroy_lock(int *mutex);
void omp_set_lock(int *mutex);
void omp_unset_lock(int *mutex);
int omp_test_lock(int *mutex);
```

Figure 25 The OpenMP lock routines.

```
void omp_set_critical_lock();
void omp_unset_critical_lock();
void omp_barrier();
void omp_set_runtime_lock();
void omp_unset_runtime_lock();
```

Figure 26 Routines used to implement CRITICAL, BARRIER, DO, SECTIONS, and SINGLE directives.

The implementation of the synchronization routines is generally very simple as the routines only need to invoke POSIX routines with similar functionality. For example, upon invocation, the *omp_init_lock* routine creates an instance of the *pthread_mutex_t* type. This instance is then initialized through a call to the *pthread_mutex_init* routine, and finally assigned to the *mutex* pointer.

3.2.2.3 The Scheduling Routines

The scheduling group of routines is used for the scheduling of iterations of the parallel DO loops. The routine prototypes are listed in Figure 27.

```
void f_omp_init_sched(int *type, int *init, int *limit, int *step,
                      int *chunk, int *thread_num);
int f_omp_get_sched_vars(int *init, int *limit, int *thread_num);
```

Figure 27 The scheduling routines.

The *f_omp_init_sched* routine is used to store the information about the type of scheduling, loop bounds, and chunk size. This routine is invoked prior to entering parallel DO loops.

The *f_omp_get_sched_vars* routine is called by each thread in order to obtain the next set of iterations it needs to execute. The calculation of the iteration set is dependent on the scheduling type. In the case of static scheduling, it is calculated as follows:

```
new init = loop init+(thread_no+iteration_no*no_of_threads)*chunk*step
```

```
new limit = new init+(chunk-1)*step
```

Dynamic scheduling requires a slightly different calculation:

```
new limit = old init + step*(chunk-1)
new init = new limit + step
```

And finally, guided scheduling uses the following formulas:

```
new chunk = [(old limit-old init)/step+1]/no_of_threads
new limit = old init + step*(new chunk-1)
new init = new limit + step
```

3.2.2.4 The OpenMP Environment routines

The OpenMP environment routines are used to control and query the parallel execution environment. Their implementation is very straightforward as they only set or retrieve certain properties of the run-time system. For example, the *omp_in_parallel* routine tells the caller if it is called within a dynamic extent of a region executing in parallel. The implementation of this routine only needs to retrieve the number of threads executing at the calling time. If this number is greater than one, the routine returns true, otherwise it returns false. The full list of the OpenMP environment routines is listed in Figure 28.

```
void omp_set_num_threads(int *num);
int omp_get_num_threads();
int omp_get_max_threads();
int omp_get_num_procs();
int omp_in_parallel();
void omp_set_dynamic(int *expr);
int omp_get_dynamic();
void omp_set_nested(int *expr);
int omp_get_nested();
```

Figure 28 The OpenMP environment routines.

3.2.2.5 The run-time state objects

The run-time state objects are a group of run-time library variables that store the information about the state of the run-time system. In particular, they keep track of the live threads, point to shared variables passed as parameters, store the scheduling information, and provide means for synchronization. Figure 29 shows some of the more significant run-time objects.

```
/* Thread information */
int num_of_threads;
pthread_t *threads;

/* Parallel subroutine */
void *(*g_f)();

/* Shared variables passed as parameters */
long *g_parameters;

/* Scheduling information */
static struct {
    int type;
    int init;
    int limit;
    int step;
    int orig_chunk;
    int chunk;
    int valid;
    int *static_rotation;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} schedule;

/* Synchronization information */
pthread_mutex_t runtime_mutex;
pthread_mutex_t critical_mutex;
struct {
    pthread_cond_t cond;
    pthread_mutex_t mutex;
} barrier;
```

Figure 29 The OpenMP environment routines.

Chapter 4

Experimental evaluation

This chapter evaluates the correctness, performance, and overhead of the code generated by our source-to-source compiler. It also provides a comparison between our compiler on the one hand, and another commercial OpenMP compiler on the other. The comparison is made in terms of the overhead incurred in using different OpenMP directives. In addition, the chapter looks at the performance of the generated code, and discusses speedup results obtained for sample applications.

The rest of the chapter is structured as follows. Section 4.1 deals with the correctness of the implementation of the compiler, and describes the methodology used for its testing. Section 4.2 evaluates the speedup performance of our compiler, and compares the overhead of the code generated by our compiler to that of other compilers.

4.1 Testing Requirements

The testing of the source-to-source compiler is divided into two areas: testing of the language extensions (i.e. front end), and functional testing (i.e. back end). Both of these areas are described in more detail in the following sections.

4.1.1 Front end testing

Front end testing ensures not only that the source-to-source compiler accepts all OpenMP language extensions, but also that it saves all relevant information about those extensions. This is accomplished by inspecting the code created by compiling various OpenMP enriched FORTRAN programs. In order to ensure that the OpenMP directives are correctly stored, the back end pass is temporarily disabled so that no code transformation occurs. The compiler is then used to produce the same code with the same OpenMP directives as the input program. The only acceptable

differences are when a clause is defined multiple times (where allowed), and when parallel work-sharing constructs are split. In the first case, our compiler will combine all occurrences of the same clauses, and in the second, the parallel work-sharing construct (such as PARALLEL DO) will be split into a parallel region construct (PARALLEL), followed by a work-sharing construct (DO).

The test programs are divided into two groups: programs using a single OpenMP directive with various clauses, and programs combining an assortment of directives and clauses. The first group is used to verify that for every directive, the compiler accepts all allowable combinations of clauses, as well as multiple instances of the same clauses with different arguments. The second group is used to test the acceptance of all applicable combinations of directives within the same program. The compiler successfully passed all tests in both groups.

4.1.2 Back end testing

Back end testing ensures that FORTRAN programs using OpenMP directives behave within the specifications of the OpenMP FORTRAN Application Interface v.1.0. Testing is done in three steps. First, a test program is compiled with the source-to-source compiler. Second, the parallel code obtained from the first step is compiled with a g77 compiler[2] and linked with our OpenMP run-time library. Finally, the executable code is run in parallel on up to 4 processors, and its output is compared to the expected output for this test. The test is deemed correct when the output of the final executable program matches the expected output.

Back end tests are divided into three groups: test programs that contain a single OpenMP directive and their respective clauses, test programs that use parallel work-sharing constructs, and test programs that combine multiple directives and various clauses. A sample of the test programs can be found in Appendix B. The compiler was found to generate correct code for all test programs in all three test groups.

4.1.3 Coverage

The front end testing covers all OpenMP directives and environment routines. All tests were performed correctly by the compiler, which successfully accepted and properly stored all

[2] GNU g77 version 2.7.2

OpenMP language extensions. Semantic error checking was not tested due to the fact that it is not implemented within the front end. This is an area that can be improved in the future.

The back end testing covers all but two OpenMP directives: OMP FLUSH, and OMP THREADPRIVATE, and two OpenMP clauses: COPYIN, and ORDERED. These directives and clauses were not tested, since they are not implemented and the back end currently ignores them. All other directives and their clauses were successfully tested and the compiler was found to produce correct code transformations.

In addition to the front, and back end testing that was done with a predefined set of programs, the compiler was also tested using the synthetic benchmarks, and the EPCC's OpenMP Microbenchmarks [6] applications. These programs were successfully parallelized and executed, thus validating the source-to-source compiler transformations on some real world applications. These applications are discussed in more detail in section 4.2.

4.2 Benchmarks

In order to evaluate the performance of the source-to-source compiler as accurately as possible, a set of standard benchmarks was used to measure the speed and overhead of the generated code. The benchmarks were first compiled with the source-to-source compiler to generate parallel code. The resulting parallel FORTRAN programs were then compiled and linked with the OpenMP run-time into executable objects. The FORTRAN compiler used to generate the executable code was GNU's g77 v.2.7.2. The executable code was then run on a multiprocessor machine whose configuration is shown in Table 2. All runs were performed in a dedicated mode were the benchmark application had an exclusive access to all of machine's resources.

Machine name	System model	Number of CPUs	CPU speed in MHz	Main memory	OS
Bastet	Sun Ultra Enterprise 450-BA Model 4400	4	400	2.0 GB	SunOS 5.5.1

Table 2. Machine configurations.

This process allowed to benchmark both the overhead incurred by the implementation of various OpenMP constructs, and the speedup of the applications parallelized using the source-to-

source compiler. The following sections present the analysis and results obtained for the speedup and overhead benchmarks.

4.2.1 Speedup benchmarks

Speedup is the ratio of the execution time of a sequential program to the execution time of the parallel version of the original program. The results presented in this section are speedups of two OpenMP programs. The first one 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 an 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). This program uses Jacobi's method to approximate the solution of the Laplace Finite Difference Equation. The benchmark is used with matrix sizes of 1000 × 1000 and 5000 × 5000 along with the STATIC scheduling scheme.

4.2.1.1 The Synthetic benchmark

Table 3 shows two sets of speedups of the synthetic benchmark on *Bastet*. The first set shows the speedup with respect to an OpenMP version of the original program running only one thread (scalability speedup), and the second one presents the speedup calculated with respect to the original program without OpenMP directives (true speedup). The first set is used to show the improvements in speed as the number of processors is increased, where the second set provides an insight into how the parallelization costs affect the speed improvements. The curve marked as STATIC (OPTIMAL) is obtained by using the STATIC scheduling type without specifying the chunk size.

Speedup type	Number of CPUs	STATIC (OPTIMAL)	DYNAMIC (chunk=1000)	GUIDED
Scalability speedup	1	1.000	1.000	1.000
	2	2.002	1.994	1.982
	3	3.001	2.542	3.037
	4	3.987	3.347	3.964
True speedup	1	1.000	1.000	1.000
	2	2.000	1.743	1.798
	3	2.998	2.222	2.756
	4	3.982	2.926	3.598

Table 3 Synthetic benchmark speedups.

Figure 30, and Figure 31 show the synthetic benchmark speedup as a function of the number of processors.

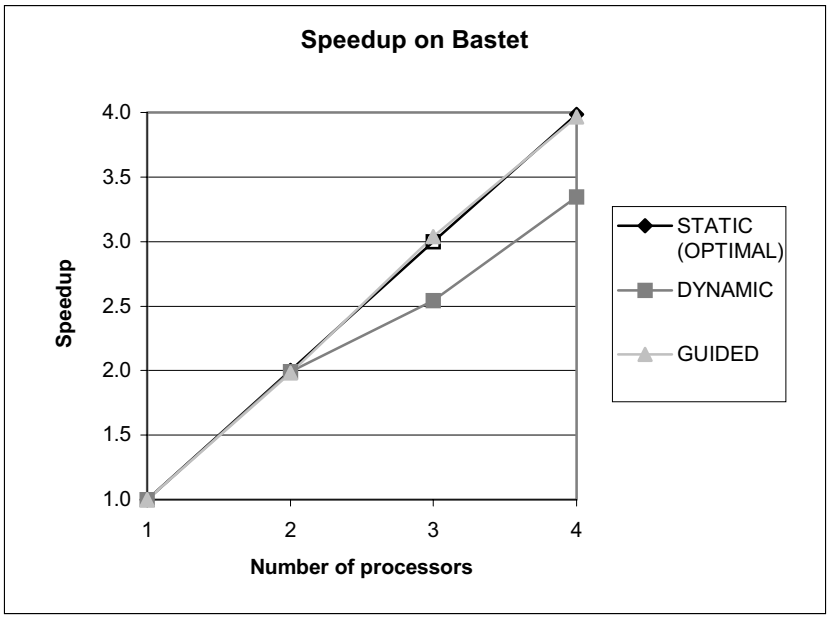

Figure 30 Scalability speedup graph for the synthetic benchmark.

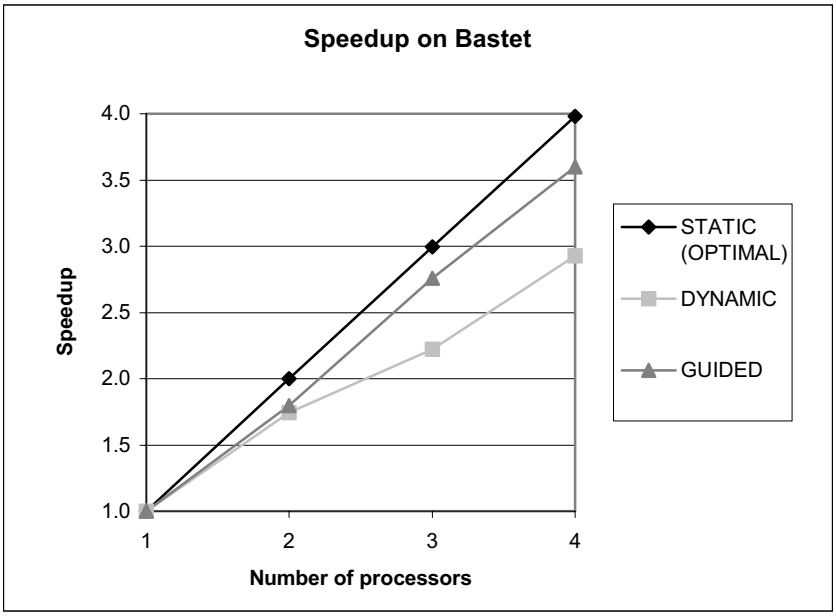

Figure 31 True speedup graph for the synthetic benchmark.

The results of the synthetic benchmarks show almost perfect scalability speedups when the scheduling method is either static, or guided. On the other hand, dynamic scheduling speedup shows some run-time communication overhead. This is attributed to the fact that when a thread finishes a piece of iteration space, it has to request a new set of iterations from the OpenMP run-time system. This communication coupled with the time spent in the run-time library routine, creates an overhead that becomes apparent as the number of requests grows. In general, the smaller the chunk size used for dynamic scheduling, the more requests are made to the run-time library, and therefore the bigger the overhead the program will incur.

The true speedup graph uncovers an additional amount of overhead. It shows that when the speedup is calculated with respect to the original program, as opposed to an OpenMP version running only one thread, the slopes of the speedup lines slightly drop. This overhead is a direct result from the thread creation and run-time set-up costs that are associated with an initialization of a multithreaded application. The graph also exposes the differences between the static and guided scheduling methods. Although very similar on the scalability speedup graph, the guided method does not perform as well as the static one when using true speedup measurements. This is not unexpected since OpenMP source-to-source compiler implementation requires guided scheduling to communicate with the run-time system in order to receive the next piece of the iteration space that needs to be executed.

4.2.1.2 The Jacobi benchmark

Table 4 shows two sets of speedups of the Jacobi benchmark on *Bastet*. The benchmark uses the STATIC (OPTIMAL) scheduling scheme. As with the case of the synthetic benchmark, the first set shows the true speedup, and the second one presents the scalability speedup. The results of the benchmarks are presented for both 1000 × 1000 and 5000 × 5000 matrices.

Speedup type	Number of CPUs	1000 × 1000 speedups	5000 × 5000 speedups
Scalability speedup	1	1.000	1.000
	2	1.990	1.989
	3	2.983	2.968
	4	3.920	3.910
True speedup	1	1.000	1.000
	2	1.951	1.946
	3	2.924	2.903
	4	3.843	3.825

Table 4 Jacobi benchmark speedups.

Figure 32, and Figure 33 show the Jacobi speedups for the matrix size 1000 × 1000 only, as both sets of results produced very similar speedups.

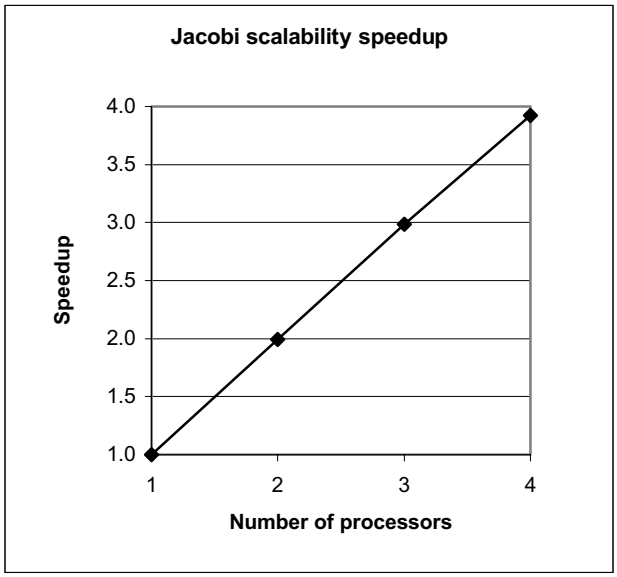

Figure 32 Scalability speedup graph for the Jacobi benchmark.

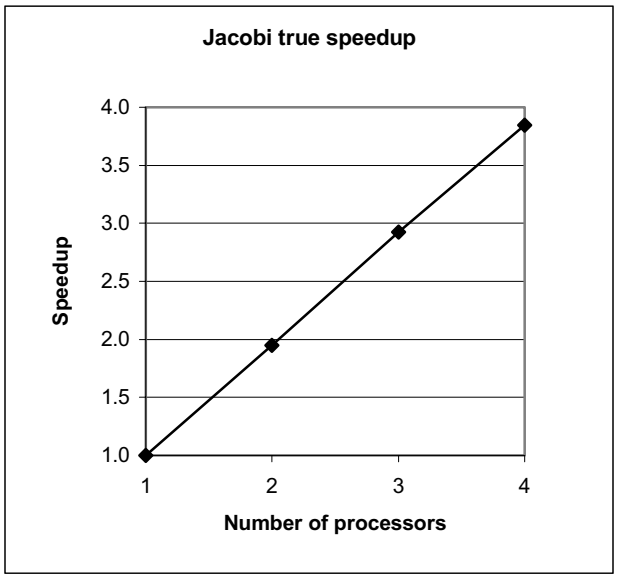

Figure 33 True speedup graph for the Jacobi benchmark.

The results of the Jacobi benchmark show near perfect scalability speedup. The true speedup graph on the other hand, reveals a very small amount of overhead that is associated with

the initialization of a multithreaded application. This overhead is the same overhead encountered by the synthetic benchmark.

4.2.2 Overhead benchmarks

In order for the compiler to schedule loops and synchronize threads, it needs to insert scheduling and synchronization calls into the source code. These calls then invoke appropriate run-time routines, which are responsible for initialization, synchronization, and scheduling issues. Hence, the execution of these run-time library routines introduces loop scheduling and synchronization overheads that may degrade the performance of the resulting parallel programs. Consequently, it becomes extremely important to minimize the amount of overhead introduced to maintain good performance.

To measure these classes of overhead for language constructs in OpenMP, Edinburgh Parallel Computing Centre (EPCC) has designed a set of micro-benchmarks that can give compiler designers an insight into problematic areas. 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 are used to compare with the overhead of the source-to-source compiler implementation.

The following sections present the results of the EPCC benchmarks using the source-to-source compiler, as well as published results of a commercial OpenMP compiler: version 3.7 of Guide F90 by Kuck and Associates (KAI). The EPCC generated results for KAI compiler on a 400-Mhz Sun E3500 machine, and presented them in processor cycles. Since the processor architecture, and the clock rate for this machine is identical to those of *Bastet*, it is possible to make a meaningful comparison of the two sets of results.

4.2.2.1 BARRIER Synchronization overhead

Barrier synchronization overhead measured by the EPCC micro-benchmarks results from the use of PARALLEL, DO, PARALLEL DO, BARRIER, and SINGLE directives. These directives, as dictated by the OpenMP specifications, synchronize the code regions they encompass, unless otherwise specified. The EPCC micro-benchmarks define barrier overhead as follows: if T_s is the sequential time for a section of code, and T_p the time for the parallel version of this code on p processors, the overhead is given by $O_p = T_p - T_s/p$. For example, to measure

the overhead of the PARALLEL directive, they subtract the times taken for the execution of the code shown in Figure 34, and Figure 35, and divide them with the number of repetitions.

```
      do i=1,reps
         call delay(delay_length)
      end do
```

Figure 34 Sequential code.

```
      do i=1,reps
!$OMP PARALLEL
         call delay(delay_length)
!$OMP END PARALLEL
      end do
```

Figure 35 Parallel code.

The delay routine call shown above, invokes a routine that contains only a dummy loop of length delaylength. EPCC chose the lengths of the dummy and repeat loops to give sensible times which can be subtracted without loss of too much precision or accuracy. In addition, multiple measurements are taken within a run, and multiple runs are executed and the times averaged. Table 5 shows the benchmark results in processor cycles[3] measured on *Bastet*.

Machine name	Processor speed in MHz	Number of processors	PARALLEL	DO	PARALLEL DO	PARALLEL + REDUCTION	BARRIER	SINGLE
Bastet	400	0	0	0	0	0	0	0
		1	17368	572	17984	18812	348	216
		2	31416	13000	32688	33120	11892	3204
		3	47192	25156	48992	48640	23164	6096
		4	62952	37560	65364	65872	34360	9064

Table 5 Barrier synchronization overhead timings in processor cycles.

Figure 36 and Figure 37 show the barrier synchronization overhead as a function of the number of processors, for the source-to-source and KAI compiler.

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

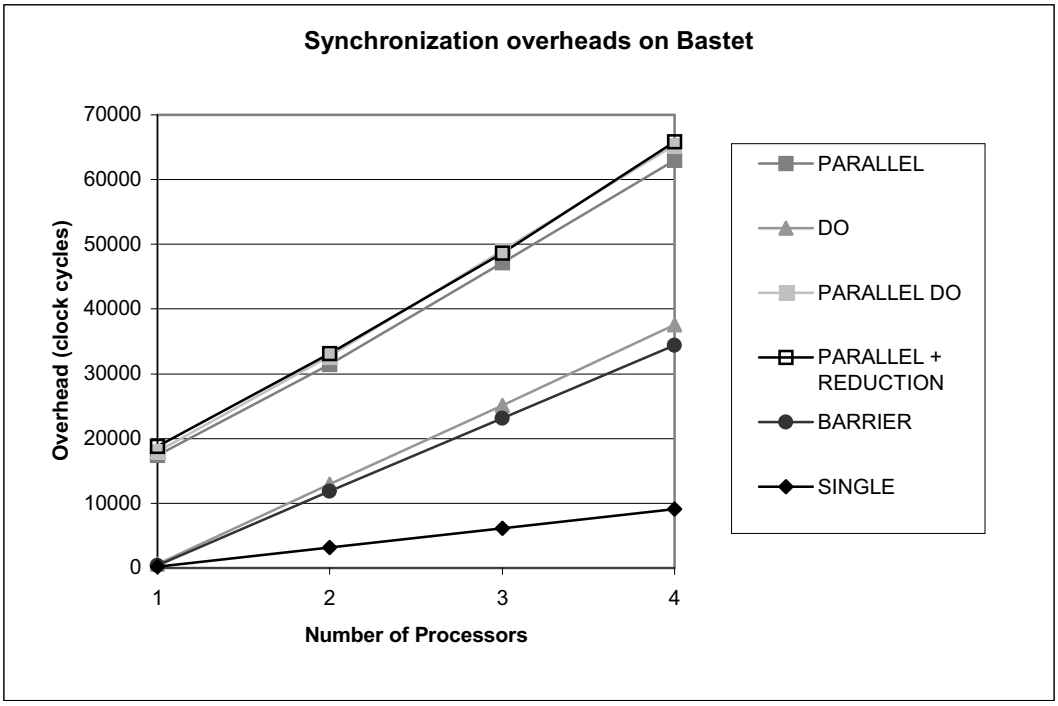

Figure 36 Source-to-source barrier synchronization graph.

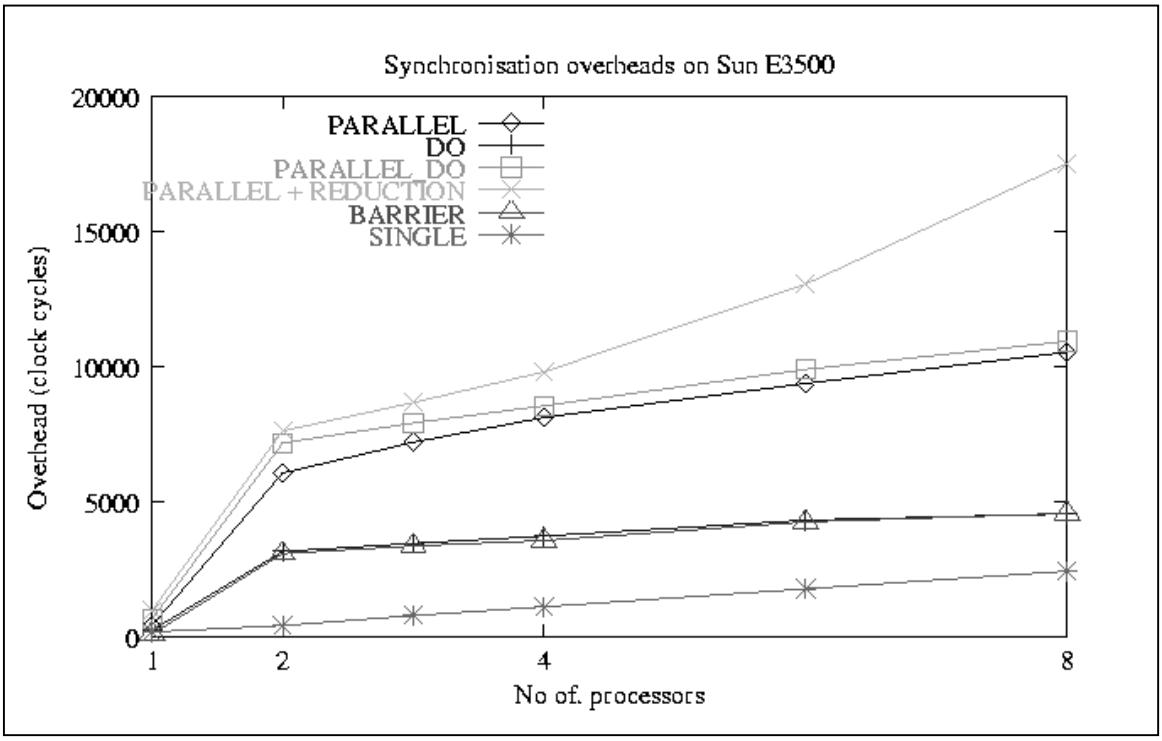

Figure 37 KAI barrier synchronization graph.

Figure 36 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 the source-to-source compiler uses the underlying implementation of the POSIX library, which does not provide scalable barrier synchronization. Nonetheless, the amount of overhead is small.

When the source-to-source compiler and KAI's compiler overhead graphs are compared, the source-to-source generated code shows a slightly steeper slope, resulting in approximately 2-3 times more overhead than KAI compiled code. This can be attributed to two factors:

- Currently, our OpenMP runtime library does not implement thread pooling. In the case where parallel regions are repeatedly created, this impedes performance.
- As previously mentioned, OpenMP runtime library depends on the POSIX threads library. Therefore, application performance will heavily depend on the underlying thread implementation. The benchmark results suggest that KAI's implementation of the threading mechanism outperforms its equivalent within the POSIX threads library.

4.2.2.2 Costs of the mutual exclusion mechanisms

The mutual exclusion benchmark exposes the cost of the mutual exclusion mechanism resulting from the use of CRITICAL, ATOMIC, LOCK, and UNLOCK directives. The costs are measured in a similar matter to the barrier synchronization overhead measurements. For example, the benchmark measures the overhead of the CRITICAL directive by subtracting the execution time for code in Figure 38 from the reference time, and then dividing this number by the number of repetitions.

```
!$OMP PARALLEL
      do j=1,reps/omp_get_num_threads()
!$OMP CRITICAL
         call delay(delay_length)
!$OMP END CRITICAL
      end do
!$OMP END PARALLEL
```

Figure 38 Parallel code.

Table 6 shows the benchmark results measured on *Bastet*.

Machine name	Processor speed in MHz	Number of processors	CRITICAL	LOCK UNLOCK	ATOMIC
Bastet	400	1	492	556	152
		2	424	540	156
		3	388	596	152
		4	532	596	152

Table 6 Mutual exclusion overhead timings.

Figure 39, and Figure 40 show the mutual exclusion overhead as a function of the number of processors, for the source-to-source and KAI compilers respectively.

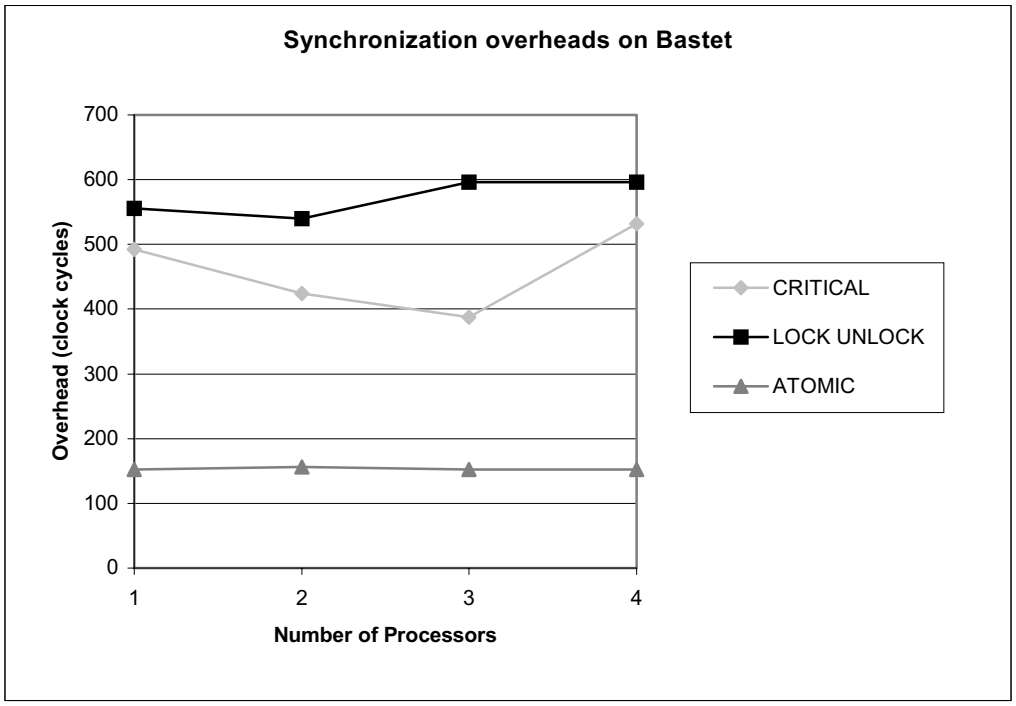

Figure 39 Source-to-source mutual exclusion overhead graph.

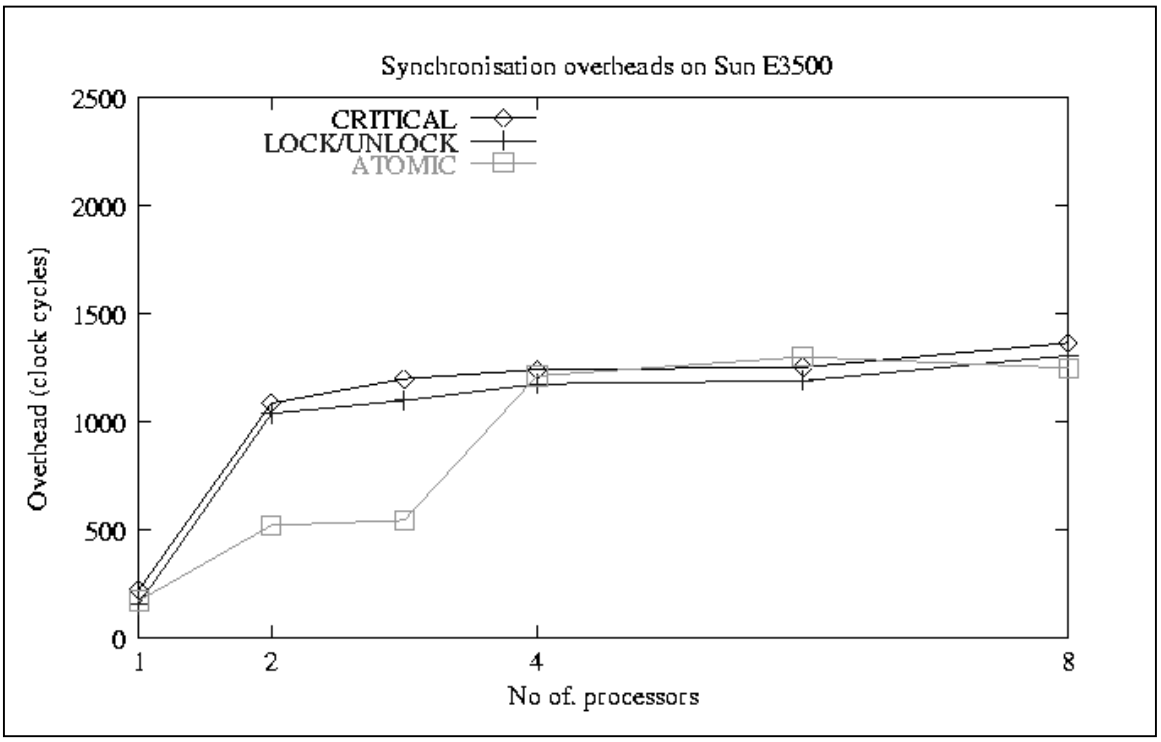

Figure 40 KAI mutual exclusion overhead graph.

Figure 39 indicates that the cost of the mutual exclusion mechanism is not only very low, but also that it does not increase with the number of processors. The performance comparison between the two compilers also shows that the source-to-source generated code outperformed KAI compiled code by a factor of two on all tests. Since the implementation of the directives responsible for the mutual exclusion overhead is very straightforward and is solely dependent on the calls to POSIX library, it is this library's implementation of the lock/unlock routines that makes the generated overhead minimal.

4.2.2.3 Loop scheduling overhead

The loop scheduling benchmark compares the overheads of the DO directive using various scheduling options and various chunk sizes. To measure these overheads, the EPCC compares the times taken for the execution of the code shown in Figure 41, and Figure 42. The overhead calculation is then done by dividing the time taken to execute the parallel code with the number of repetitions, and subtracting this amount with the time taken to execute the sequential code divided by the number of repetitions.

```
      do j=1,reps
         do i=1,num_of_iters
            call delay(delay_length)
         end do
      end do
```

Figure 41 Sequential code.

```
!$OMP PARALLEL
      do j=1,reps
!$OMP DO SCHEDULE(sched_type,chunk_size)
         do i=1,num_of_iters*omp_get_num_threads()
            call delay(delay_length)
         end do
      end do
!$OMP END PARALLEL
```

Figure 42 Parallel code.

Table 7 shows the benchmark results measured on *Bastet* using 4 threads. The results marked as STATIC (OPTIMAL) are obtained by using the STATIC scheduling type without specifying the chunk size. This allows the compiler to divide the loop iterations among threads in contiguous pieces, where only one piece is assigned to each thread. This process minimizes the scheduling overhead as each thread executes its part of the iteration space without the need to constantly query the run-time system for the next set of iterations. The results marked as GUIDED are not measured for chunk sizes greater than 32 since these sizes would not allow the GUIDED scheduling scheme to employ all 4 threads simultaneously. This is due to the length of the inner loop (`num_of_iters = 128`), and the methodology of calculating the size of the iteration space that is to be dispatched (`iteration size = number of iterations left / number of threads`). For example, if the chunk size were 64, then the calculated iteration size (32) would be smaller than the minimum size allowed (64), and the whole iteration space would have to be assigned to one thread.

Machine name	Processor speed in MHz	Chunk size	STATIC	DYNAMIC	GUIDED	STATIC (OPTIMAL)
Bastet	400	1	87184	1128032	177976	44360
		2	69572	675228	173120	
		4	56780	411976	168092	
		8	50396	269284	165580	
		16	47808	182132	157440	
		32	45404	132536	143016	
		64	44368	109048	-	
		128	43868	105116	-	

Table 7 Loop scheduling overhead timings.

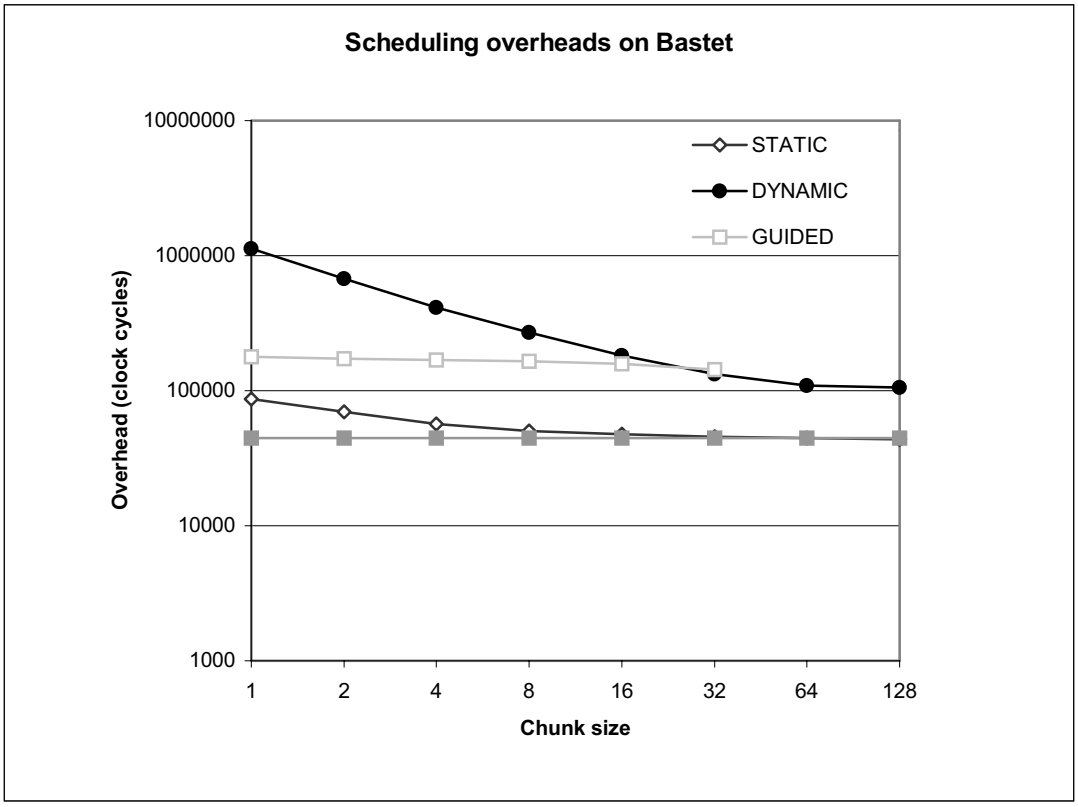

Figure 43 Source-to-source loop scheduling overhead graph.

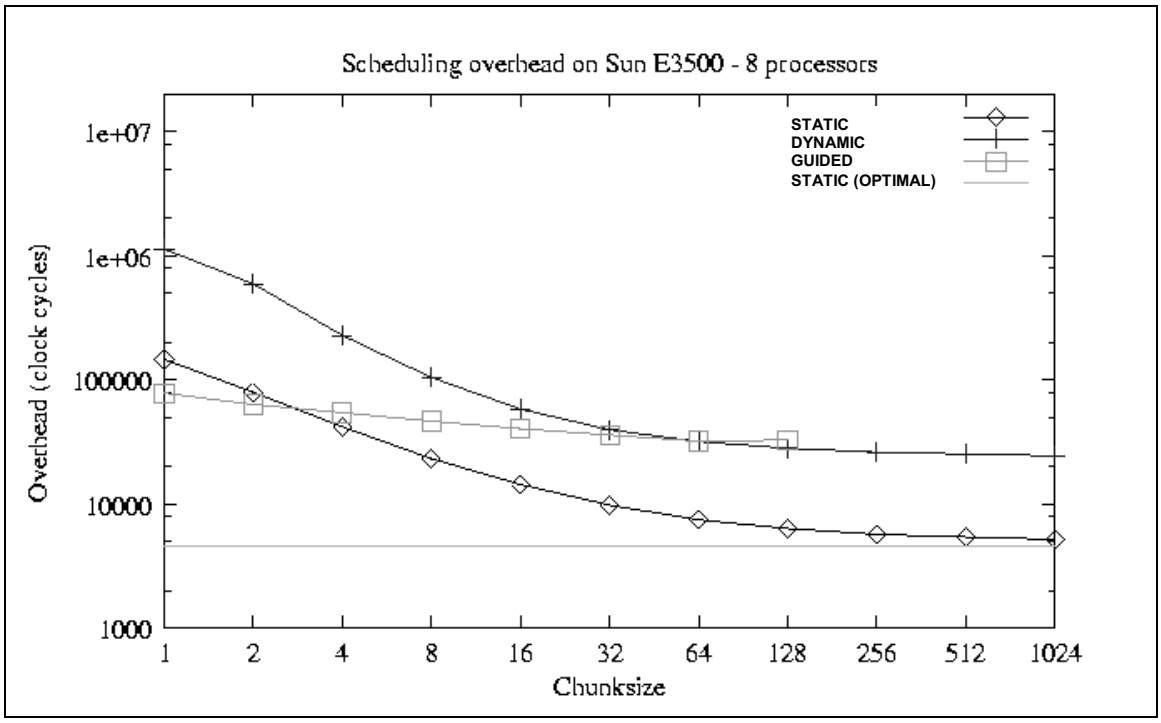

Figure 44 KAI loop scheduling overhead graph.

Figure 43, and Figure 44 show the loop scheduling overhead curves for the code generated by the source-to-source and KAI compilers respectively. The figures indicate that although both sets of curves exhibit almost identical behaviour, the source-to-source compiler curves are positioned higher on the graph. This difference can be easily seen when looking at the STATIC (OPTIMAL) scheduling results. The source-to-source generated code produces approximately 44000 cycles of overhead, where the KAI code generates 5000 cycles. The difference in the amount of overhead can be explained by examining the methodology of the overhead calculation done in this benchmark. As previously shown (in Figure 42), the scheduling benchmark code contains two nested do loops. The inner loop is scheduled using the DO directive, and the outer one is used in order to be able to average out the execution times of the inner, scheduled loop. Since the benchmark measures the time that takes to execute both loops, it measures not only the scheduling overhead, but also the synchronization overheads of the PARALLEL and DO directives. The synchronization overhead of the PARALLEL directive does not have a great impact on the total amount of overhead, as it is only taken into account once. The DO directive synchronization overhead, on the other hand, affects the calculation to a greater extent. Since the inner loop is executed multiple times, the synchronization overhead of the DO directive is added to the scheduling overhead for every iteration of the outer loop. The effects of this overhead are most apparent in the case of the STATIC (OPTIMAL) scheduling. Due to its nature, this type of scheduling entails minimal amounts of overhead because there is only one iteration of scheduling involved. This implies that the overhead of the STATIC (OPTIMAL) scheduling obtained by the scheduling benchmark, is mostly attributed to the synchronization issues. The results of the synchronization benchmark confirm this presumption. Figure 36 shows that the amount of the synchronization overhead of the DO directive is almost identical to the amount of the scheduling overhead measured by this benchmark (when using 4 threads, the synchronization overhead of the DO directive is ~38000, where the overhead of the STATIC (OPTIMAL) scheduling is ~44000). This implies that the code generated by both compilers for the STATIC (OPTIMAL) type of scheduling, produces almost equivalent amount of the "real" scheduling overhead. The results obtained for the other two scheduling types suffer from the same problem. When the synchronization overhead is subtracted from the benchmark results for all three types of scheduling, both compilers show very similar amounts of the scheduling overheads.

In conclusion, the comparison of the overheads of the barrier synchronization between our compiler and KAI's, shows that our compiler generates about 2 to 3 times higher overhead.

However, our compiler's mutual exclusion overhead is smaller by a factor of 2. Lastly, both compilers produce the same amount of the loop scheduling overhead. Hence, in general, the run-time overhead of our compiler is of the same order as that of this commercial compiler.

Chapter 5

Related work

Over the last four years, OpenMP API has been implemented in a number of different compilers. Most of these implementations are extensions of the existing commercial compilers, where the compiler architecture modifications, and the underlying implementation details, are not publicly disclosed. Therefore, there is no easy way to compare the implementation approaches taken by those compiler vendors with our source-to-source compiler implementation. The implementations of OpenMP specifications that have been publicly described, and whose source code is available for inspection, are in general, research compiler projects. Some of those include RWCP Omni OpenMP Compiler [13], OdinMP/CCp [5], and NanosCompiler [2]. The following sections briefly describe them, and compare their implementations with the OpenMP source-to-source compiler implementation.

5.1 RWCP Omni OpenMP Compiler

The RWCP Omni package [13] is a collection of programs and libraries, written in C and Java, that allows researchers to build code transformation systems (compilers). The Omni OpenMP compiler is part of the software that translates OpenMP enriched C and FORTRAN 77 programs, into C code suitable for compiling with a native C compiler.

Omni's OpenMP transformation model is based on the same principle that our source-to-source compiler uses. Sequential programs annotated with parallel directives are converted into fork-join parallel programs by encapsulating the parallel regions into separate functions. At the run-time, the main program calls Omni run-time library routines to fork slave threads that then, execute the parallel function.

The main differences in the transformation model between the two compilers lie with the issues of portability, treatment of shared variables, and handling of work sharing constructs. The

Omni compiler relies on a C++ run-time library specific to an SMP cluster, therefore restricting portability to very specific platforms. In contrast, our source-to-source compiler only requires the presence of the POSIX threads library. The difference in the treatment of shared variables is a direct result of the underlying features of each of the compiler's output language. The Omni compiler generates pointers to shared variables, which are then copied into shared memory heap and passed to slaves at the fork time. The source-to-source compiler, on the other hand, creates common blocks, which are shared between the main program and the different instances of the parallel function. The two compilers also have different methods of handling work sharing constructs. In particular, the Omni compiler handles static scheduling by delegating all scheduling tasks to the run-time library, where our compiler inserts scheduling transformations directly into the parallel functions.

5.2 OdinMP/CCp

OdinMP/CCp [5] is a portable implementation of the OpenMP standard for a variety of platforms. It implements OpenMP specifications for ANSI C programming language, and produces C code that uses POSIX threads as the fundamental threading mechanism. This compiler is very similar in the functionality and underlying implementation to our compiler, but with no capability of using FORTRAN as its input language. One of the main differences in the implementation of the two compilers lies in the design of the parallelization mechanism. The OdinMP parallelization is built on top of a thread pooling feature. Its implementation dictates that all threads be created immediately upon starting the program. After their creation, the threads are suspended on a condition variable until they are called to perform work. The parallel regions are numbered and moved from their original place into a separate routine as part of one large switch statement. When a parallel region needs to be executed, a team of threads is woken up by a broadcast statement, and is given a region number that needs to be executed. Upon finishing the execution, all threads block once again until more work is available.

5.3 NanosCompiler

NanosCompiler [2] is a source-to-source parallelizing compiler implemented around a hierarchical internal program representation of Parafrase II compiler. NanosCompiler captures the parallelism expressed by the user (through OpenMP directives and extensions), and the

parallelism automatically discovered by the compiler through a detailed analysis of data and control dependences. The compiler is responsible for encapsulating work into threads, establishing their execution precedences, and selecting the mechanisms to execute them in parallel.

The goal of NanosCompiler is to provide an infrastructure for combining automatic parallelization with the parallelism provided by user annotations. This functionality, and its implementation, is significantly different from other OpenMP compilers. One of the drawbacks of an automatic parallelization compiler is that it provides very little control of the extent, and the type of the parallelization created by the compiler. NanosCompiler is not an exception despite the fact that user annotation functionality supersedes the parallelism automatically discovered by the compiler. NanosCompiler is also not a fully functional OpenMP compiler as it only implements a subset of the OpenMP specifications. In addition, it targets the Illinois-Intel multithreading library, which restricts the portability of the implementation.

Chapter 6

Conclusion

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

The compiler provides many benefits to the potential users. It provides OpenMP FORTRAN programming capability at no cost. Its portability feature allows development on different platforms while experiencing minimal environment changes required by the compiler. Lastly, and most importantly, the results of the benchmarks show that the compiler's performance is comparable with other commercially available OpenMP FORTRAN compilers, making it a viable alternative.

6.1 Future Work

There are many directions to explore with this source-to-source compiler. An obvious first step is to complete the implementation of the OpenMP FORTRAN Application Program Interface by providing a solution for the FLUSH directive for several different platforms, and extending the back end to implement the ORDERED directive. In addition, it should also be possible to augment the front end to provide semantic error checking, as well as error reporting that is based on a custom error message library. These additions will allow the compiler users to fully utilize OpenMP potential, and help resolve development issues that arise when creating and debugging parallel applications.

Future work should be also directed at ensuring full OpenMP functionality on modern versions of FORTRAN such as FORTRAN 90, and 95, as well as further improving the run-time

performance of the generated code. The improvement efforts might also be targeted towards extending the design of the OpenMP run-time library to implement thread reuse via thread pooling. This feature could reduce the run-time system overhead incurred when parallel regions are repeatedly created.

In November 2000, OpenMP API version 2.0 was released. The new release contains some directive additions and functional improvements. A possible extension of this work is to assess and incorporate the new standard into our compiler.

Chapter 7

Bibliography

[1] ANSI Technical Committee X3H5, Parallel Processing Model for High Level Programming Models, 1992.

[2] 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.

[3] 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.

[4] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Processing Technology*, 2(3): 37-47, 1994.

[5] C. Brunschen. OdinMP/CCp – A Portable Compiler for C with OpenMP to C with POSIX threads. *Master's thesis*, Lund University of Technology, 1999.

[6] J. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *European Workshop on OpenMP*, 1999.

[7] K. Faigin, J. Hoeflinger, D. Padua, P. Petersen, and S. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5): 553-586, 1994.

[8] J. Grout. Inline Expansion for the Polaris Research Compiler. *Master's thesis*, University of Illinois, 1995.

[9] K. Myeong. The Design of an Efficient and Portable Interface Between a Parallelizing Compiler and its Target Machine. *Master's thesis*, University of Illinois, 1995.

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

[11] B. Kuhn, P. Petersen, and E. O'Toole. OpenMP vs. threading in C/C++. Kuck & Associates Inc and Compaq Computer Corporation. In *European Workshop on OpenMP*, 1999.

[12] N. Manjikian. Program Transformation for Cache Locality Enhancement on Shared-memory Multiprocessors. *PhD thesis*, University of Toronto, 1997.

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

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

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

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

Appendix A

Benchmarks

A.1 Synthetic benchmark

A.1.1 speedup.f

```
      program speedup
      integer i

      write(6,*)
      write(6,10)
      write(6,11)
      write(6,10)
      write(6,*)
 10   format("###############")
 11   format("# No threads #")
      call work()

      do i=1,4
         write(6,*)
         write(6,20)
         write(6,21) i
         write(6,20)
         write(6,*)
         call OMP_SET_NUM_THREADS(i)
         write(6,22)
         call omp_work1()
         write(6,23)
         call omp_work2()
         write(6,24)
         call omp_work3()
      end do

 20   format("########################")
 21   format("# Number of threads: ", i1, " #")
 22   format("SCHEDULE(STATIC):")
 23   format("SCHEDULE(DYNAMIC,1000):")
 24   format("SCHEDULE(GUIDED):")
      stop
      end


      subroutine work()
      include 'speedup.h'
```

```
      external getclock
      real*8 getclock
      real*8 aaaa, start, end
      integer i,j,n,m,r

      n=10000
      m=10000
      aaaa=0.0
      do r=1, reps
                  start = getclock()
                  do i=1,m
                        do j=1,n
                              aaaa=aaaa+i
                        end do
                  end do
                  end = getclock()
          time(r) = end - start
      end do
      call stats()

      end


      subroutine omp_work1()
      include 'speedup.h'
      external getclock
      real*8 getclock
      real*8 aaaa, start, end
      integer i,j,n,m,r

      n=10000
      m=10000
      aaaa=0.0
      do r=1, reps
                  start = getclock()
C$OMP PARALLEL DO SCHEDULE(STATIC) PRIVATE(i,j,aaaa)
                  do i=1,m
                        do j=1,n
                              aaaa=aaaa+i
                        end do
                  end do
C$OMP END PARALLEL DO
                  end = getclock()
          time(r) = end - start
      end do
      call stats()

      return
      end



      subroutine omp_work2()
      include 'speedup.h'
      external getclock
      real*8 getclock
      real*8 aaaa, start, end
      integer i,j,n,m,r

      n=10000
      m=10000
      aaaa=0.0
      do r=1, reps
```

```
                start = getclock()
C$OMP PARALLEL DO SCHEDULE(DYNAMIC,1000) PRIVATE(i,j,aaaa)
                do i=1,m
                       do j=1,n
                              aaaa=aaaa+i
                       end do
                end do
C$OMP END PARALLEL DO
                end = getclock()
            time(r) = end - start
        end do
        call stats()

        return
        end



        subroutine omp_work3()
        include 'speedup.h'
        external getclock
        real*8 getclock
        real*8 aaaa, start, end
        integer i,j,n,m,r

        n=10000
        m=10000
        aaaa=0.0
        do r=1, reps
                start = getclock()
C$OMP PARALLEL DO SCHEDULE(GUIDED) PRIVATE(i,j,aaaa)
                do i=1,m
                       do j=1,n
                              aaaa=aaaa+i
                       end do
                end do
C$OMP END PARALLEL DO
                end = getclock()
            time(r) = end - start
        end do
        call stats()

        return
        end



        subroutine stats ()
        include 'speedup.h'
        real*8 meantime, totaltime, sumsq, mintime, maxtime, sd
        mintime = 1.0e20
        maxtime = 0.
        totaltime = 0.

        do i = 1, reps
           mintime = min(mintime,time(i))
           maxtime = max(maxtime,time(i))
           totaltime = totaltime + time(i)
        end do

        meantime  = totaltime/ real(reps)

        sumsq = 0.
```

```
      do i = 1, reps
         sumsq = sumsq + (time(i)-meantime)**2
      end do

      sd = sqrt(sumsq/(reps-1))

      cutoff = 3.0 * sd

      nr = 0
      do i = 1, reps
         if (abs(time(i)-meantime) .gt. cutoff) nr = nr + 1
      end do

c
      write (6,1000)
      write (6,*)
      write (6,1001)
      write (6,1002) reps, meantime, mintime, maxtime, sd, nr
      write (6,*)
      write (6,1003) meantime, conf95*sd
      write (6,*)
      write (6,1000)
 1000 format("------------------------------------------------------------
     $--------------");
 1001 format("Sample_size        Average    Min        Max        S.D.
     $      Outliers")
 1002 format(4x,i7,4x,4f11.5,4x,i7)
 1003 format("Ran in ", f11.5, " seconds +/- ", f11.5)
      return
      end
```

A.1.2 speedup.h

```
      integer reps
      parameter (reps=5)
      real*8 time(0:reps), conf95
      parameter (conf95 = 1.96)
      common /data/ time
```

A.1.3 getclock.c

```
#include <sys/time.h>
#include <unistd.h>

double getclock_()
{
        double t;
        struct timeval tv;
        struct timezone tz;
        gettimeofday(&tv, &tz);
        t = (double) tv.tv_sec + (double) tv.tv_usec * 1.0e-6;
        return t;
}
```

A.2 Jacobi benchmark

A.2.1 jacobi.f

```
      program main
*************************************************************
* program to solve a finite difference
* discretization of Helmholtz equation :
* (d2/dx2)u + (d2/dy2)u - alpha u = f
* using Jacobi iterative method.
*
* Modified: Mario Soukup, 2001
*           Sanjiv Shah,       Kuck and Associates, Inc. (KAI), 1998
* Author:   Joseph Robicheaux, Kuck and Associates, Inc. (KAI), 1998
*
* Directives are used in this code to achieve paralleism.
* All do loops are parallized with default 'static' scheduling.
*
* Input :  n - grid dimension in x direction
*          m - grid dimension in y direction
*          alpha - Helmholtz constant (always greater than 0.0)
*          tol   - error tolerance for iterative solver
*          relax - Successice over relaxation parameter
*          mits  - Maximum iterations for iterative solver
*
* On output
*       : u(n,m) - Dependent variable (solutions)
*       : f(n,m) - Right hand side function
*
* Working varaibles/arrays
*     dx  - grid spacing in x direction
*     dy  - grid spacing in y direction
*
*************************************************************
      implicit none

*      include 'jacobi.h'
      integer n,m,mits
      double precision tol,relax,alpha
      parameter (n = 5000)
      parameter (m = 5000)
      parameter (alpha = 5.0)
      parameter (relax = 1.0)
      parameter (tol = 0.0000001)
      parameter (mits = 1000)
      double precision u(n,m),f(n,m),dx,dy

      integer i,j, xx,yy
      double precision PI
      parameter (PI=3.1415926)

      integer k,k_local
      double precision error,error2,resid,rsum,ax,ay,b
      double precision error_local, uold(n,m)
      real ta,tb,tc,td,te,ta1,ta2,tb1,tb2,tc1,tc2,td1,td2
      real te1,te2
      real second
      external second

      double precision xxd,yyd,temp
```

```
***************
* Timing data *
***************
      integer r, reps
      parameter (reps=10)
      real*8 time(0:reps), conf95
      parameter (conf95 = 1.96)
      common /data/ time
      external getclock
      real*8 getclock, start, end

****************
* Start timing *
****************
      do r=1, reps
         start = getclock()


*****************************************************
* Initializes data
* Assumes exact solution is u(x,y) = (1-x^2)*(1-y^2)
*
*****************************************************

      dx = 2.0 / (n-1)
      dy = 2.0 / (m-1)

* Initilize initial condition and RHS

c$omp parallel do private(xx,yy)
      do j = 1,m
         do i = 1,n
            xx = -1.0 + dx * dble(i-1)        ! -1 < x < 1
            yy = -1.0 + dy * dble(j-1)        ! -1 < y < 1
            u(i,j) = 0.0
            f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy)
     &           - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy)
         enddo
      enddo
c$omp end parallel do


*****************************************************************
* Solve Helmholtz equation :
* Solves poisson equation on rectangular grid assuming :
* (1) Uniform discretization in each direction, and
* (2) Dirichlect boundary conditions
*
* Jacobi method is used in this routine
*
* Input : n,m   Number of grid points in the X/Y directions
*         dx,dy Grid spacing in the X/Y directions
*         alpha Helmholtz eqn. coefficient
*         relax Relaxation factor
*         f(n,m) Right hand side function
*         u(n,m) Dependent variable/Solution
*         tol    Tolerance for iterative solver
*         mits  Maximum number of iterations
*
* Output : u(n,m) - Solution
*****************************************************************
```

```
*
* Initialize coefficients
      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b  = -2.0/(dx*dx)-2.0/(dy*dy) - alpha ! Central coeff

      error = 10.0 * tol
      k = 1

      do while (k.le.mits .and. error.gt. tol)

         error = 0.0

* Copy new solution into old
c$omp parallel

c$omp do
         do j=1,m
            do i=1,n
               uold(i,j) = u(i,j)
            enddo
         enddo

* Compute stencil, residual, & update

c$omp do private(resid) reduction(+:error)
         do j = 2,m-1
            do i = 2,n-1
*     Evaluate residual
               resid = (ax*(uold(i-1,j) + uold(i+1,j))
     &                + ay*(uold(i,j-1) + uold(i,j+1))
     &                 + b * uold(i,j) - f(i,j))/b
* Update solution
               u(i,j) = uold(i,j) - relax * resid
* Accumulate residual error
               error = error + resid*resid
            end do
         enddo
c$omp end do nowait

c$omp end parallel

* Error check

         k = k + 1
         error = sqrt(error)/dble(n*m)
*
      enddo                     ! End iteration loop
*

************************************************************
* Checks error between numerical and exact solution
*
************************************************************

      dx = 2.0 / (n-1)
      dy = 2.0 / (m-1)
      error2 = 0.0

c$omp parallel do private(xxd,yyd,temp) reduction(+:error2)
      do j = 1,m
         do i = 1,n
```

```
            xxd = -1.0d0 + dx * dble(i-1)
            yyd = -1.0d0 + dy * dble(j-1)
            temp  = u(i,j) - (1.0-xxd*xxd)*(1.0-yyd*yyd)
            error2 = error2 + temp*temp
         enddo
      enddo
      error2 = sqrt(error2)/dble(n*m)



*****************
* Finish timing *
*****************

          end = getclock()
          time(r) = end - start
      end do
      call stats()

      print *, 'Total Number of Iterations ', k
      print *, 'Residual                   ', error
      print *, 'Solution Error : ',error2

      stop
      end


      subroutine stats ()
      integer i, reps
      parameter (reps=10)
      real*8 time(0:reps), conf95
      parameter (conf95 = 1.96)
      common /data/ time
      real*8 meantime, totaltime, sumsq, mintime, maxtime, sd
      mintime = 1.0e20
      maxtime = 0.
      totaltime = 0.

      do i = 1, reps
         mintime = min(mintime,time(i))
         maxtime = max(maxtime,time(i))
         totaltime = totaltime + time(i)
      end do

      meantime  = totaltime/ real(reps)

      sumsq = 0.
      do i = 1, reps
         sumsq = sumsq + (time(i)-meantime)**2
      end do

      sd = sqrt(sumsq/(reps-1))

      cutoff = 3.0 * sd

      nr = 0
      do i = 1, reps
         if (abs(time(i)-meantime) .gt. cutoff) nr = nr + 1
      end do

c
      write (6,1000)
      write (6,*)
```

```
      write (6,1001)
      write (6,1002) reps, meantime, mintime, maxtime, sd, nr
      write (6,*)
      write (6,1003) meantime, conf95*sd
      write (6,*)
      write (6,1000)
 1000 format("------------------------------------------------------------
     $--------------");
 1001 format("Sample_size        Average    Min        Max        S.D.
     $      Outliers")
 1002 format(4x,i7,4x,4f11.5,4x,i7)
 1003 format("Ran in ", f11.5, " seconds +/- ", f11.5)
      return
      end
```

A.3 The EPCC Microbenchmarks

The EPCC Microbenchmarks can be found on the EPCC Microbenchmark page:

http://www.epcc.ed.ac.uk/research/openmpbench/index.html

Appendix B

Testing programs

B.1 Sample of front-end test programs

B.1.1 front_end_1.f

```
      program test
      integer a(100,100), b
      common /vars/ a,b
      b = 5
c$OMP PARALLEL DEFAULT(NONE) SHARED(a) PRIVATE(b)
      do i = 1, 100
         a(i,i) = b
      end do
c$OMP END PARALLEL
      print *, a(100,100)
      end
```

B.1.2 front_end_2.f

```
      program test
      integer a(100,100), b
      common /vars/ a,b
      b = 5
c$OMP PARALLEL DEFAULT(NONE) SHARED(a,b)
c$OMP DO SCHEDULE(STATIC,11) REDUCTION(+:b)
      do i = 1, 100
         b = b+1
         a(i,i) = b
      end do
c$OMP END PARALLEL
      print *, a(100,100), b
      end
```

B.2 Sample of back-end test programs

B.2.1 back_end_1.f

```
      program test
      integer*8 i,j
      print *, "Started"
      j = 0
c$OMP PARALLEL REDUCTION(+:j)
c$OMP DO SCHEDULE(DYNAMIC, 5)
      do i=16, 30002, 3
        j = j+i
      end do
c$OMP END PARALLEL
      print *, "Finished: j =", j
      end
```

B.2.2 back_end_1.expected

```
Started
Finished: j = 150024966
```

B.2.3 back_end_2.f

```
      program test
      integer*8 i, i1, i2, i3, j, j1, j2, j3
      print *, "Started"
      i = 5
      i1 = 6
      i2 = 7
      i3 = 8
c$OMP PARALLEL

c$OMP SECTIONS
      j = i+1
c$OMP END SECTIONS

c$OMP SECTIONS
c$OMP SECTION
      j = i+2
c$OMP END SECTIONS

c$OMP SECTIONS
      j = i+3
c$OMP SECTION
c$OMP END SECTIONS

c$OMP SECTIONS
c$OMP SECTION
      j = i+4
c$OMP SECTION
c$OMP END SECTIONS

c$OMP SECTIONS
```

```
c$OMP SECTION
      j = i+5
c$OMP SECTION
      j1 = i+5
      j1 = j1+i1
c$OMP SECTION
      j2 = i+5
      j2 = j2+i1
      j2 = j2+i2
c$OMP SECTION
      j3 = i+5
      j3 = j3+i1
      j3 = j3+i2
      j3 = j3+i3
c$OMP END SECTIONS

c$OMP END PARALLEL
      print *, "Finished: ", i, i1, i2, i3, j, j1, j2, j3
      end
```

B.2.4 back_end_2.expected

```
Started
Finished:  5 6 7 8 10 16 23 31
```