OdinMP/CCp - A portable implementation of OpenMP for C

Christian Brunschen and Mats Brorsson

Department of Information Technology, Lund University
P.O. Box 118, SE-221 00 Lund, Sweden

Mats.Brorsson@it.lth.se

Abstract

We describe here the design and performance of OdinMP/CCp which is a portable compiler for C-programs using the OpenMP directives for parallel processing with shared memory. OdinMP/CCp was written in Java for portability reasons and takes a C-program with OpenMP directives and produces a C-program for POSIX threads.

We describe some of the ideas behind the design of OdinMP/CCp and show some performance results achieved on an SGI Origin 2000 and a Sun E10000. Speedup measurements relative a sequential version of the test programs show that OpenMP programs using OdinMP/CCp exhibits excellent performance on the Sun E10000 and reasonable performance on the Origin 2000.

1. Introduction

Parallel architectures that support a shared address space in hardware have now become commonplace and they are no longer confined to just a few processors. Servers such as the Sun E10000 and the SGI Origin 2000 are often equipped with 64 processors or more. Unfortunately, until quite recently, each manufacturer of shared memory architectures has had its own way of writing programs with a shared address space model. With the advent of OpenMP, a new industry standard for a shared address space programming model, this has changed [2].

The OpenMP standard API is specified for Fortran and C/C++ [3, 4]. The API consists of a set of compiler directives to express parallelism, work sharing, data storage and synchronisation. In addition there are a number of library routines that can be used for synchronisation and to control the work sharing in more detail. The directives extend the programming language but does not change it. Compilers that do not understand the OpenMP directives can safely ignore them.

OpenMP implementations now exist for most shared memory parallel platforms. However, up until very recently all compilers that understand OpenMP directives are either vendor propriatary, such as the MIPSpro compiler from SGI, or commercial, such as the compilers from Kuck&Associates and Portland Group. There is, however, a need for portable public domain implementations in order to let users of parallel computers try out OpenMP before being committed financially.

In this paper we describe the design, implementation and performance of a portable implementation of the OpenMP specification for C. The result is a C-to-C translator written in Java that takes a C program with OpenMP directives and produces a C program using POSIX threads, or pthreads for short, to implement the parallelism. The resulting translator implements the entire specification for sequentially consistent shared memory multiprocessors using pthreads. For multiprocessors with a relaxed memory model, a slight modification to the compiler is needed for strictly correct behaviour for volatile variables.

We first approached the problem by attempting to develop a mechanical way to manually translate an OpenMP program into an equivalent program using pthreads. In this process, we had to solve a number of conceptual problems. Pthreads offer a granularity of parallelism at the level of one function, which is quite radically different from what OpenMP offers. Likewise, we had to investigate, understand and solve such problems as thread initialization, how to implement threadprivate variables, or how to handle the fact that source code can come in more than one file. Also, while the main focus was to investigate the viability of doing this at all, we could not entirely lose sight of performance issues, both regarding memory use and overhead introduced into the code by the translation.

To write the translator itself, we chose to perform the development in Java, with the help of two compiler-writing tools, Java Tree Builder and Java Compiler Compiler [5, 6]. The result is a working compiler called OdinMP/CCp (CCp stands for C-to-C with pthreads). It generates code that offers good performance, and it has the advantages that is can be used on any platform that offers support for POSIX threads.

Preliminary performance measurements on an SGI Origin 2000 and a Sun E10000 shows that the performance of OdinMP/CCp is reasonable even comparing with a commercial OpenMP implementation, except in certain circumstances.

OdinMP/CCp is publicly available with complete source code at the following URL: http://www.it.lth.se/odinmp. It is also currently being used in another project, to convert the SPLASH program suite from ANL macros to OpenMP, in order to compare the two approaches to parallel programming of shared memory architectures.

2. Design of the translator

We will in this section give an overview of how OdinMP/CCp translates a C-program with OpenMP directives to a pthreads program. A more detailed description of both the design and the implementation can be found in [1].

2.1 Creating parallelism

The only way to create parallel activity in OpenMP is by the means of the parallel directive as shown below:

```
#pragma omp parallel
{
   /* Code to be executed in parallel */
}
```

This is in sharp contrast to pthreads where a new parallel thread is specified by designating a function for the new thread to run. This means that anything we want to be able to run in parallel in the OpenMP program must be inside this function or called by it. The basic way that OdinMP/CCp deals with this issue works as follows:

1. OdinMP defines a function `thread_function`, which basically is a large, initially empty, switch statement.
2. Each parallel region in the program is assigned a unique identifying number. The code inside the parallel region is moved from its original place in the program, into the switch statement inside `thread_function`, where it can be selected by its associated number.
3. OdinMP/CCp defines a function `thread_spinner`, which waits for work, and calls `thread_function` to perform the actual task. This function is the basic function each thread executes.

The parallel construct is replaced with code that:

- Allocates a team of threads.
- Tells each thread in the team to execute the parallel region in question.
- Runs the parallel region itself as the master thread of the team, and
- Waits for all the other threads in the team to finish.

This is illustrated in figure 1.

2.2 Work sharing constructs

The `for`-construct

The most common work sharing construct in OpenMP is the `for`-construct. This constructs divides the iterations in a `for`-loop among the available threads. We show below a simple example in which a `for`-loop with 100 iterations is parallelised. It is assumed that this code is executed inside a parallel region. Otherwise it is ignored.

```
#pragma omp for
for (i = 0; i < 100; i++){
   /* parallel code */
}
```

The OpenMP `for`-construct distributes the iterations of a `for`-loop into smaller slices and hands them to different threads which then runs the slices in parallel. To do this in OdinMP/CCp we first extract the range of the `for`-loop from the loop header. The each thread performs the following repetitively.

- Fetch a slice of the shared loop.
- If there are no more slices then this thread is done with the `for`-loop and exits the `for`-construct.
- Otherwise, iterate over the iterations as defined by the fetched slice.

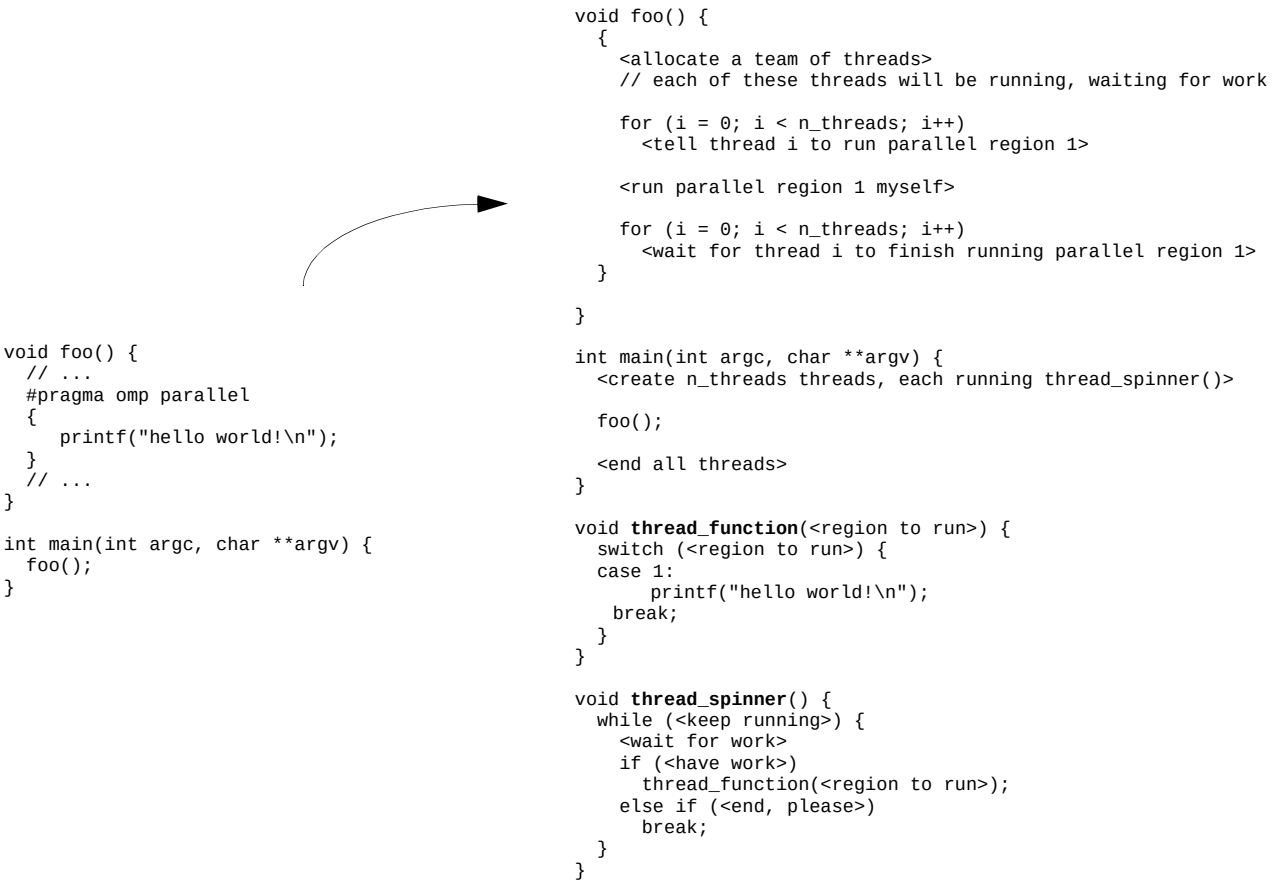
This is illustrated in figure 2. The arrows indicate how OdinMP/CCp extracts information from the `for`-loop header to find the loop index variable, the loop initialisation value, the boundary value and the increment.

The `sections` construct

The `for`-construct is useful to distributed work in loop parallelism. Function parallelism can be expressed by means of the `sections` construct. In the example below functions `foo()` and `bar()` are executed in parallel.

```
#pragma omp parallel
{
#pragma omp sections
   {
#pragma omp section
   { foo(); }
#pragma omp section
   { bar(); }
   }
}
```

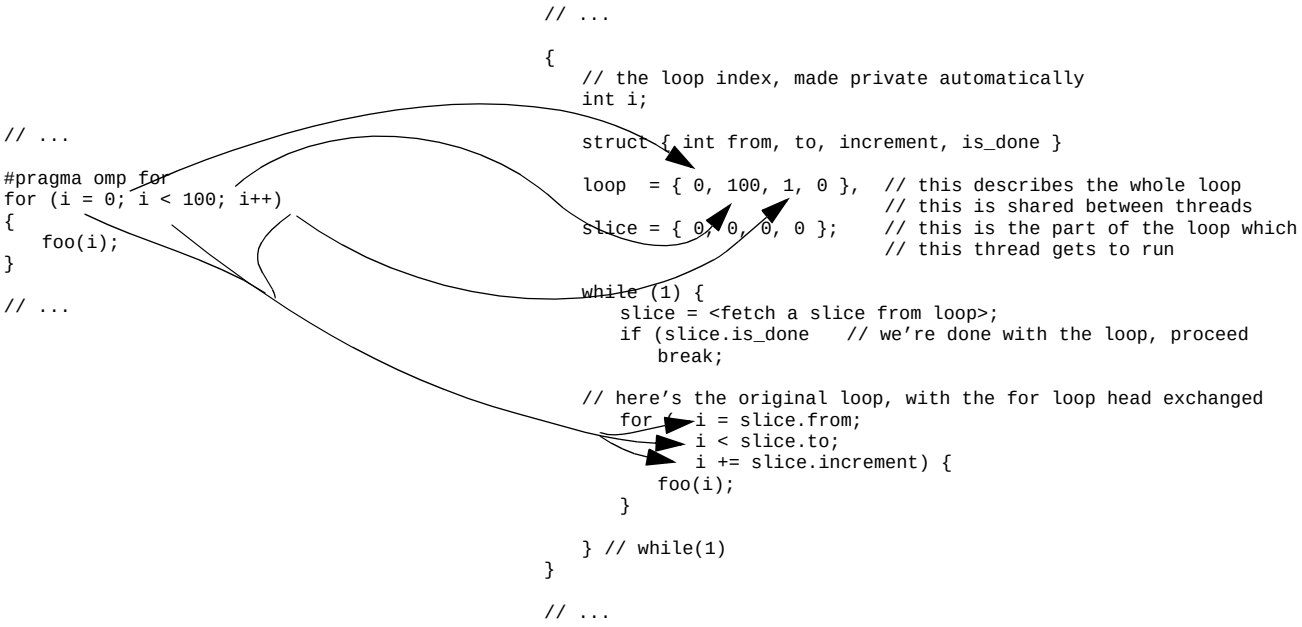
The way this is solved in OdinMP/CCp is similar to the way slices of iterations are handed out to threads for a `for`-loop. We collect information about the number of sections to run and give each section a unique identity. Each thread then performs the following:



```
void foo() {
  // ...
  #pragma omp parallel
  {
     printf("hello world!\n");
  }
  // ...
}

int main(int argc, char **argv) {
  foo();
}
```

```
void foo() {
  {
    <allocate a team of threads>
    // each of these threads will be running, waiting for work

    for (i = 0; i < n_threads; i++)
      <tell thread i to run parallel region 1>

    <run parallel region 1 myself>

    for (i = 0; i < n_threads; i++)
      <wait for thread i to finish running parallel region 1>
  }

}

int main(int argc, char **argv) {
  <create n_threads threads, each running thread_spinner()>

  foo();

  <end all threads>
}

void thread_function(<region to run>) {
  switch (<region to run>) {
  case 1:
       printf("hello world!\n");
   break;
  }
}

void thread_spinner() {
  while (<keep running>) {
    <wait for work>
    if (<have work>)
      thread_function(<region to run>);
    else if (<end, please>)
      break;
  }
}
```

Figure 1. The principle of the translation of a simple OpenMP program (left) to pthreads (right).



```
// ...

#pragma omp for
for (i = 0; i < 100; i++)
{
   foo(i);
}

// ...
```

```
// ...

{
   // the loop index, made private automatically
   int i;

   struct { int from, to, increment, is_done }

   loop  = { 0, 100, 1, 0 },  // this describes the whole loop
                              // this is shared between threads
   slice = { 0, 0, 0, 0 };    // this is the part of the loop which
                              // this thread gets to run

   while (1) {
      slice = <fetch a slice from loop>;
      if (slice.is_done   // we're done with the loop, proceed
         break;

   // here's the original loop, with the for loop head exchanged
      for (i = slice.from;
           i < slice.to;
           i += slice.increment) {
         foo(i);
      }

   } // while(1)
}

// ...
```

Figure 2. The translation of the `for`-construct in OpenMP

- Fetch a slice of sections, i.e., fetch the number of sections this thread is to run.
- If there is no slice available, the thread is ready and exits the sections code.
- Otherwise, it executes the appropriate sections.

2.3 Data scope attributes

A parallel region and/or a work sharing construct can have a number of data scope attributes that control the way variables are accessed. By default, all variables visible in a parallel region are shared among the threads unless they have been allocated with automatic storage, e.g. on the stack, within the dynamic extent of a parallel region. Please refer to the OpenMP specification for a more detailed description of the data environment [3].

OpenMP provides a number of directives to change the default behaviour. The following example gives a flavour of the data scope attributes:

```
static int s1;

void foo() {
   int s2, p, fp, rdx;
   /* ... */
#pragma omp parallel shared(s1,s2) \
                     private(p) \
                     firstprivate(fp) \
                     reduction(+ : rdx)
   {
      bar();
   }
   /* ... */
}
```

Here there are five different variables that have to be dealt with differently. Variables `s1` and `s2` are declared as shared. This would not have been necessary as they are visible from within the parallel region and therefore are shared by default but it is good practice to explicitly declare them as shared. Variable `s1` is a global static variable and is normally allocated in the data section of the program. This section is shared among all threads in a pthreads program and we therefore do not need to take any special action for this variable.

Variable `s2`, however, is declared within the lexical context of function `foo()` and is therefore allocated on the stack. Since the stack is private to each thread we have to change all references to `s2` to a shared memory area.

The private variable, `p`, in this example is re-declared in the thread-function (see figure 1) as well as the firstprivate variable `fp`. However, since `fp` should be initialised from the master thread's copy a shared version of it is also declared in order to communicate the initialisation value.

Finally, the reduction variable is declared in a special data structure for each thread. Each copy of the reduction variable is initialised according to the reduction operation and all copies are later combined into the original reduction variable.

3. Implementation

OdinMP/CCp is written in Java based on JavaCC [6] and the Java Tree Builder [5]. JavaCC – The Java Compiler Compiler – distributed by Metamata Inc., is a simple yet powerful LL(1) parser generator and was suitable as a starting point as it includes a complete grammar for ANSI C.

The Java Tree Builder from Purdue University is a preprocessor for JavaCC. It takes a simple JavaCC grammar and processes it, generating classes to describe each non-terminal node in the grammar and rewriting the grammar so that the resulting parser will build a tree of nodes corresponding to the parsed data.

The grammar used was the sample C grammar distributed with JavaCC with additions for the OpenMP directives and constructs.

The choice of Java to implement OdinMP/CCP has lead to a portable binary distribution of the translator. The translator has been tested on a generic dual processor PC with Linux, an SGI Origin 2000 and various Sun multiprocessors and was found to be portable across these platforms without any change in the distribution at all. The only platform dependent code concerns the way how pthreads are scheduled. In Solaris and Linux it is possible to specify that POSIX threads should contend for CPU resources in the same way as processes. This is, however, not possible without super-user privileges in IRIX, the operating system used on the SGI Origin. Therefore, an additional call is added on IRIX platforms in order to provide a hint to IRIX that it should schedule the threads in parallel. Unfortunately this scheduling policy has a negative impact on the performance on IRIX platforms.

4. Performance

4.1 Overview

In order to utilise pthreads to implement OpenMP programs a lot of extra code has to be added to the application. This creates extra overheads and it is therefore not self-evident that OdinMP/CCp as described here would be useful to write production code. We have done a preliminary performance evaluation in which we have run a set of five applications and measured the resulting speedup.

4.2 Experimental setup

Platforms

We have used two different platforms for our experiments. The first platform is an SGI Origin 2000 with a total of 100 300 MHz MIPS R12000 processors at LUNARC, Lund university. The other platform is a Sun E10000 with a total of 64 250 MHz UltraSPARC processors.

Both systems implement POSIX threads although somewhat differently. On the Sun machine it is possible to specify that a POSIX thread should be scheduled together with processes by specifying a system scope for scheduling. This is equivalent to bind a thread to a light-weight process using Solaris threads. On the SGI machine, this operation is reserved for super-users. Instead a call is made to the threads library to suggest the ideal number of processors that this program is to be run on. It is, however, up to the operating system to decide the number of processors during run-time.

On the Sun E10000 we have only used OdinMP/CCp but on the SGI Origin 2000 we have compared the performance of OdinMP/CCp with that of MIPSpro ver. 7.3 which has support for OpenMP for C. MIPSpro 7.3 was also used as the back-end compiler when using OdinMP/CCp. On the Sun E10000 we used gcc as back-end compiler. All compilations were done with the optimisation level -O3.

Applications

We have used five different applications in the performance evaluation:

- *pi* – a calculation of pi using a simple series. 100 million partial sums.
- *md* – a simple molecular dynamics simulation. This is a C-version of the md-program available from the OpenMP web site and developed by Kuck&Associates. 2048 molecules and 10 time steps.
- *laplace* – solving the laplace equations with an explicit method. 1000 by 1000 elements in the matrix and iterating 100 times.
- *cg* – solving an unstructured sparse linear system with a conjugate gradient method. This is an OpenMP version of the same benchmark in the NAS parallel benchmark suite. Using the class A parameters.
- *cg-orphan* – solving the same problem as cg but using orphaned OpenMP directives with only one parallel region instead of a parallel region for each work sharing construct.

4.3 Speedup measurements

Figures 3 to 7 show the speedup using for the five applications. The speedups are relative to the sequential execution time without using OpenMP directives on each platform. We have executed each application five times for each configuration and used the average of the measured execution times. I should be noted that the sequential execution times on the Sun E10000 is approximately three times as long as the sequential execution times on the SGI Origin 2000.

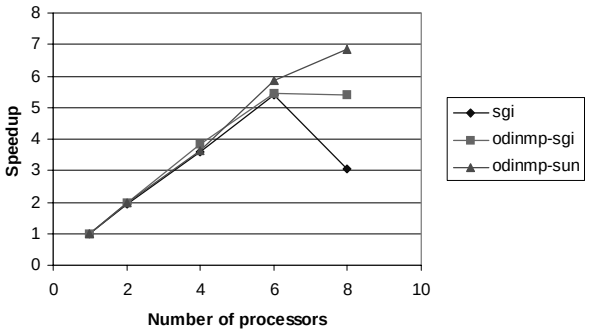


Figure 3. Speedup for *pi*.

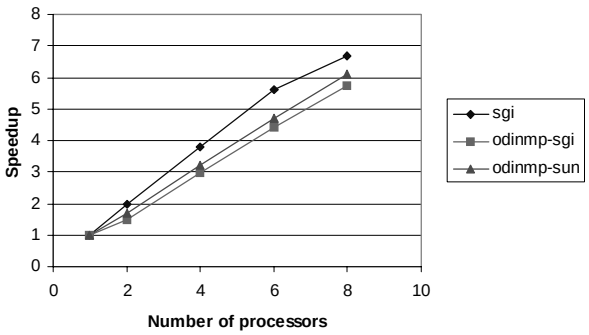


Figure 4. Speedup for *md*.

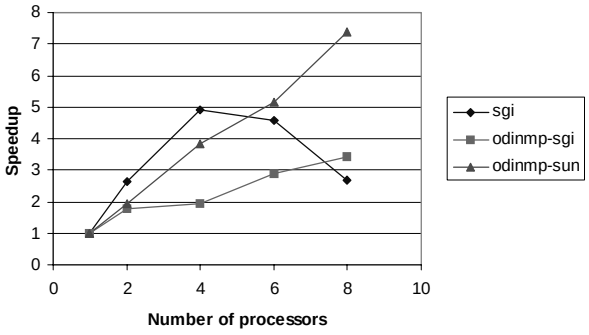


Figure 5. Speedup for *laplace*.

The speedups were measured on 2 to 8 number of threads. The actual parallelism achieved is a function of the

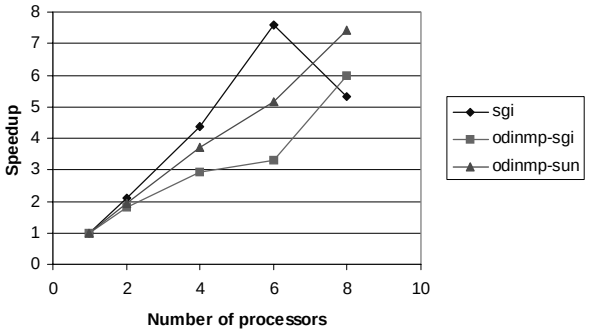


Figure 6. Speedup for *cg*.

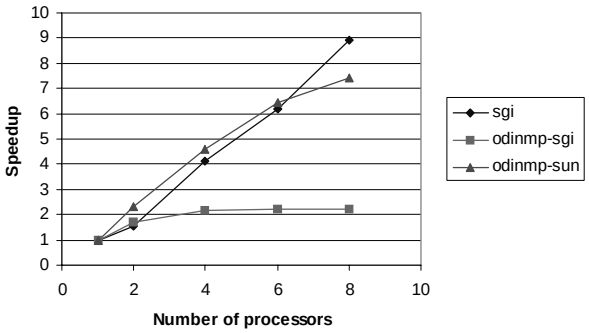


Figure 7. Speedup for *cg-orphan*.

application parallelism and how the operating system schedules the threads on available processors.

From the figures we can see that OdinMP/CCp on the Sun E10000 performs similarly as the MIPSpro OpenMP compiler on the SGI Origin 2000. It is, however, unclear how the difference in scalar performance affects the speedup.

OdinMP/CCp performs reasonably well also on the SGI Origin 2000. The performance numbers are, however, preliminary as we have so far been unable to make a truly fair comparison between Odin MP/CCp which uses pthreads and the MIPSpro compiler which uses the SGI proprietary sprocs light-weight processes. The measurements have been carried out on a loaded system and the standard deviations of the execution times were high.

We have so far been unable to explain why OdinMP/CCp performs badly for cg-orphan on the Origin 2000 while it performs well on the Sun E10000.

5. Conclusions

We are encouraged by the fact that it has been possible in a relatively short period of time – six months to be precise – to develop an implementation of the OpenMP standard for C. OdinMP/CCp has some limitations. It only works for ANSI C and it requires the underlying architecture to be sequentially consistent as it ignores the `flush` directive.

One major goal when OdinMP/CCp was developed was portability. This has been achieved thanks to the POSIX threads interface and the fact that the translator was written in Java. Even if the Java byte-code is interpreted the performance of the actual translation process is not poor.

Finally we are also encouraged by the performance of OpenMP. Even though we have only performed a preliminary performance evaluation the results so far indicates that it has a reasonable performance. We intend to characterise the performance more thoroughly in the future and to gradually improve it.

Acknowledgements

We gratefully acknowledge the use of the computing resources of LUNARC, centre for scientific and technical computing at Lund University and UNICC, Unix Numeric Intensive Calculations at Chalmers.

References

[1] Christian Brunschen, *OdinMP/CCp – A Portable Compiler for C with OpenMP to C with POSIX threads*, MSc thesis, Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden, July 1999.

[2] OpenMP Architecture Review Board, *OpenMP: A Proposed Standard API for Shared Memory Programming*, White paper, http://www.openmp.org.

[3] OpenMP Architecture Review Board, *OpenMP C and C++ Application Pogram Interface*, Version 1.0, October 1998. http://www.openmp.org

[4] OpenMP Architecture Review Board, *OpenMP Fortran Application Pogram Interface*, Version 1.0, October 1997. http://www.openmp.org

[5] Jens Palsberg, Kevin Tao and Wanjun Wang, *The Java Tree Builder*. http://www.cs.purdue.edu/jtb

[6] Sriram Sankar, Sreenivasa Viswandha, Rob Duncan and Juei Chang, *JavaCC, The Java Compiler Compiler.* http://www.metamata.com/JavaCC