OdinMP/CCp – A Portable Compiler for C with OpenMP to C with POSIX threads

Christian Brunschen

LUND INSTITUTE OF TECHNOLOGY
Lund University

Department of Information Technology

OdinMP/CCp

A Portable Compiler

for C with OpenMP

to C with POSIX threads

Master of Science Thesis

Christian Brunschen

1999-07-23

Department of Information Technology, Lund University

Abstract

The subject of my Master Thesis in Computer Science has been to write a compiler which implements the OpenMP specification for C, and generates C code using POSIX threads (a.k.a. pthreads) to implement the parallelism.

I first approached the problems by attempting to develop a mechanical way to manually translate an OpenMP program into an equivalent program using pthreads. In this process, I 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, I 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, I could not entirely lose sight of performance issues, both regarding memory use and overhead introduced into the code by the translation.

Though performed manually, the resulting scheme for translating was quite mechanic and suitable for implementation in software. My next task, thus, was to write a compiler for C with OpenMP, which would generate much the same code as I had generated manually. This compiler should also be reasonably portable, and still generate code which would run with reasonable performance. To write the compiler itself, I succumbed to the lure of Java, for which a number of good supporting tools are available for the purpose of writing compilers.

The result is a working compiler called OdinMP/CCp, which implements all the mandatory parts of the OpenMP specification. It generates code that offers quite reasonable performance, and it has the advantages that is can be used on for platform that offers support for POSIX threads.

1 Introduction

The purpose of this article is to present and discuss the OdinMP/CCp subproject of the OdinMP project, which aims to create a freely available, portable set of implementations of the OpenMP [1] standard for parallel computing, for a variety of different platform/language combinations.

Parallel computing offers a solution for problems which are too large for a single processor to handle: divide the problem into parts, let one processor handle each part concurrently – so that the parts are solved in parallel – and combine the part-results into a result for the whole problem.

OpenMP is a set of directives which, when added to sequentially written code (code which is written to run on only one processor), instructs the compiler how to divide the problem into parts to prepare it for parallel processing.

In this article, I will give a brief overview of parallel computing in general, describe the basic concepts, problems and their most common solutions. Likewise, I will present a cursory look at the prevalent programming models for parallel computer systems.

I will then discuss OpenMP, giving both a general introduction and some more detailed examples. Following this, I will detail how one can manually parallelize a sequentially written program in a semi-mechanical way, according to OpenMP primitives.

The main part of this article will describe the automatic compiler, which is the main result of this thesis work. It takes standard ANSI C [2] language code, annotated with OpenMP directives, and generates a corresponding program in which all OpenMP constructs have been replaced with equivalent code in ANSI C using the POSIX threads library [3], also known as 'pthreads'. I will cover the tools that I have used, design decisions and my reasons for making them, problems that I have stumbled over along the way and their solutions, and of course the end result.

Finally, I will present performance and overhead measurements for code as generated by OdinMP/CCp, and compare it to a commercial OpenMP implementation.

My main focus in designing and implementing OdinMP/CCp has been completeness and correctness rather than performance, especially for the translation process itself, though I have taken reasonable care not to introduce any unnecessary performance overhead in the resulting program. Alas, the speed of the OdinMP/CCp compiler itself is less than stunning, perhaps in some part due to the choice of Java as the implementation language. However, the resulting programs run well and correctly, and exhibit performance and speedup consistent with that generated in the manual parallelization process also described in this article. This speedup trails somewhat behind what platform-dependant compilers can generate, but this is not surprising, given that the OdinMP/CCp compiler generates C source code as output and thus does not have access to whatever 'shortcuts' might be possible on the underlying architecture.

2 Parallel Computing

I will present a cursory look at some of the basic concepts in parallel computing. More detailed information is available in many places, for instance [4] and [5].

2.1 *Quick Overview*

A simple model of a computer can be sketched something like this: A processing unit (CPU) which is connected to a memory unit, usually some sort of secondary storage, and a number of input/output devices – keyboard, mouse, monitor, network interface, etc.

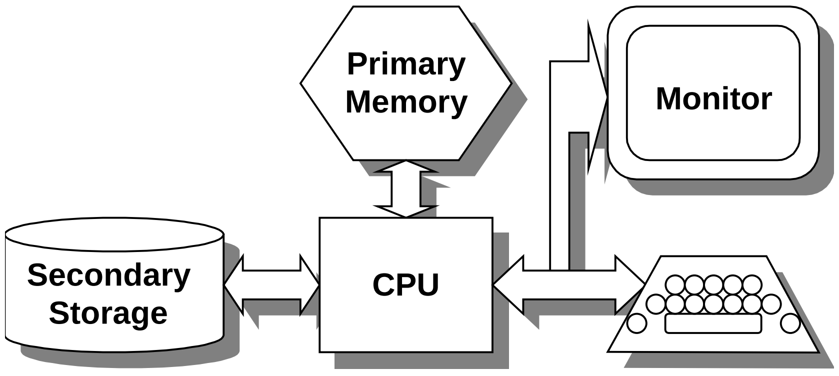


Figure 1, Simple Model of a Computer System

The function of the CPU can be roughly described as reading a series of commands from memory, and acting on those instructions one at a time. This means that the overall performance of the system is limited by the speed at which the CPU can process this series of instructions. One way to increase the system's performance is, obviously, to increase the speed at which the CPU operates – and this is indeed one of the things that is constantly happening. However, we often need more power than a single CPU can supply on its own. One solution to this problem that immediately pops up is that of trying to let several CPUs share the work – letting each work on a piece of the whole problem simultaneously with the others. This solution is generally referred to as *parallel computing* or *multiprocessing,* because instead of the computer operating on only one series of instructions, the computer now has several CPUs – multiple processors – each executing a series of instructions in *parallel* with the others.

However, one important distinction can be drawn between parallel systems where all processors work on the *same* instruction stream – such that all processors perform the same actions, but on different data – and those where each processor works on its own distinct series of instruction. These two concepts are called *Single Instruction, Multiple Data* (*SIMD*) and *Multiple Instruction, Multiple Data* (*MIMD*), respectively.

The *SIMD* model is highly useful, but also limited to those applications where the problem can be divided into *exactly identical* parts. *MIMD* is much more flexible, since it allows different parts of the problem to be handled in different manners – indeed, different CPUs may be working on completely different parts of the problem at the same time. The *MIMD* model is thus the one that is used for general-purpose multiprocessing systems.

2.2 Programming Models

Now that we have a computer system that is capable of performing several different actions at once, of processing multiple instructions simultaneously, we need a good model for programming it – after all, our problem does not divide itself into suitable parts on its own. The basic model is one of several concurrent *processes* (or *threads*) executing simultaneously, within the context of a common computer system. One problem that needs to be addressed in such a system is that of communication between different processes, such as for the purpose of synchronization, or the exchange of data within the problem domain. Consider a simulation of planets in a solar system, where each planet's location and motion are handled by separate processes. In order to calculate the planet's behavior, each process needs access to the location data for each other planet – and thus, needs information from each other process – to calculate how the gravitational forces act between the planets. So there definitely needs to be a way to transport data between different processes on different processors. Furthermore, no process can be allowed to begin calculating the next step in the simulation before all the information it needs is available, i.e., before all other processes have finished calculating the previous step of the simulation, so we need to be able to synchronize the processes.

2.2.1 Message Passing

One model, which lends itself to consideration, is one where each processor has direct access only to its own assigned slice of information, but where each processor can send an arbitrary message to another processor.

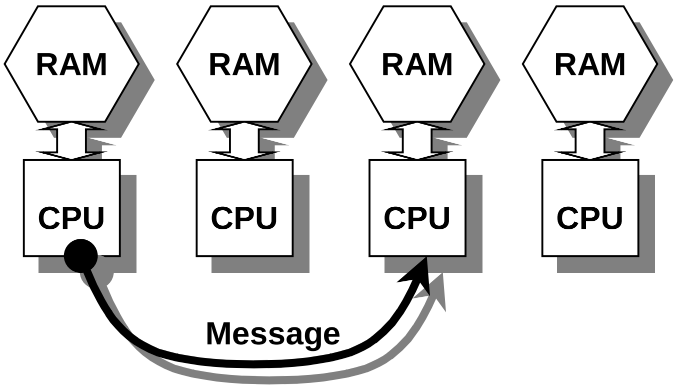


Figure 2, Message Passing

In our planetary simulation above, at the end of each step of the simulation, each process would send a message containing the information needed by the other processes for the next simulation step, to each other process. And each process can only begin calculating a new step in the simulation when it has received the messages with the current position information from each other process. So Message Passing offers both data distribution and process synchronization.

For many applications, Message Passing is a very good model. However, for problems where the same data has to be accessed by many processes, a lot of messaging ensues, which has to be handled in the program by the programmer.

A standard *Application Programmer's Interface* (*API*) for message passing exists, called *Message Passing Interface* (*MPI*) [6].

2.2.2 Shared Memory

The second prevalent model in parallel computing is one where all processors conceptually access the same, shared memory space. Communication between processes is easily achieved by simply reading from and writing to the same addresses, the same variables, in this shared memory.

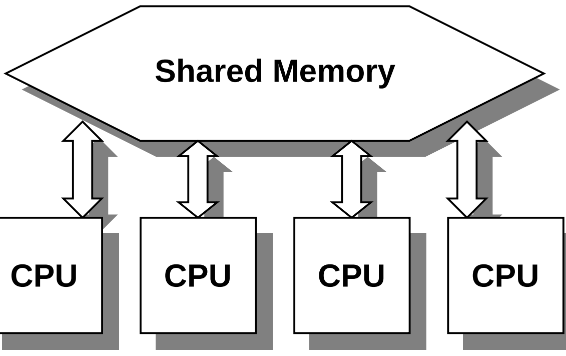


Figure 3, Shared Memory

Let us take another look at our planet simulation. Each process would have immediate access to the information calculated by each other process, simply by reading from the appropriate location in memory – the correct entry in an array, perhaps – exactly as the problem might be solved in a traditional serial program (i.e., one which runs in only one process). Through the use of a special instruction usually called 'Test and Set', specific locations in shared memory can be designated as 'locks', which in turn can be used to build higher-level primitives for synchronization. So Shared Memory also offers both data distribution and process synchronization.

3 OpenMP

3.1 Quick Overview

OpenMP [1] is a specification of a standardized set of compiler directives, which a programmer can add to the source code of an existing program, written with sequential execution in mind. The OpenMP directives allow the compiler to parallelize the code in question according to those directives, for execution on a shared-memory multiprocessor system. The OpenMP specification also includes library routines for accessing and modifying some of the execution parameters in the running program, such as the number of threads to be used. The full text of the OpenMP specification is available on the World-Wide Web, at <http://www.openmp.org/>.

3.2 Why OpenMP?

Prior to the introduction of OpenMP, there was no standard way to easily parallelize a program for use on a shared-memory multiprocessor system. While a well-entrenched standard existed for message-passing, programmers who wanted to use shared-memory systems had to either use non-standard, non-portable APIs or write cumbersome code for an underlying low-level multi-processing API, such as POSIX threads. Such programs are often excellently portable, but in the process the programmers' attention is diverted from solving the main problem, to the details of making the solution multiprocessing-capable. OpenMP changes this, solving several of the problems in this process:

- OpenMP is a widely accepted industry standard. Programmers developing to the OpenMP specification can expect their programs to be portable to a wide variety of different shared-memory multiprocessing systems, and makers of shared-memory multiprocessing systems have a well-defined API which, if they support it, makes their systems immediately viable for most of their prospective clients.
- OpenMP is non-invasive – it does not force the programmer to radically change their programming style. Programs can be written for correct sequential execution first, and

OpenMP directives and constructs can be added later, without influencing the execution of the program in the sequential case. Thus, programmers can separate the work they do on solving the actual problem from the work to parallelize the solution.
- OpenMP defines a user-guided parallelization process. This means that the compiler does not have to perform vast analysis of the code, it can and should rely *only* on the information in user-supplied directives when parallelizing the code. This gives the user complete control over what should be parallelized and how, while at the same time making the compiler much less complex.
- OpenMP is sufficiently complete. While freeing the user from the detailed labor of parallelizing the program, OpenMP still offers sufficient control over the parallelization and execution of the program, should the user have requirements above the default behavior of OpenMP.

3.3 Quick and Dirty OpenMP Tutorial

OpenMP uses the `#pragma` C compiler extension mechanism, defining an OpenMP *directive* to have the form '`#pragma omp` *directive-name* [*clause*[*clause*] …] *new-line*'. Each directive starts with `#pragma omp`, in order to reduce the risk for conflict with other pragma directives. Following that is the directive name (only one directive name can be specified in one directive), optionally followed by a number of *clauses* which affect the directive. Finally, a new-line concludes the directive.

3.3.1 `Parallel` Construct

The basic unit of parallel execution in OpenMP is the *`parallel` construct*. This consists of a *`parallel` directive* followed by a compound-statement:

```
#pragma omp parallel [clauses]
{
   /* … */
}
```

The parallel directive accepts the following clauses:

- `if(`*expression*`)`
  Run in parallel only if *expression* evaluates to a non-zero value, otherwise run sequentially
- `shared(`*list*`)`, `private(`*list*`)`, `firstprivate(`*list*`)`
  These specify whether the variables named in *list* should be shared between threads, private to each thread, or 'firstprivate'- where each thread gets a private copy, which is initialized from the current value of the corresponding variable outside the parallel region
- `copyin(`*list*`)`
  For each of the `threadprivate` (see section 3.3.7 for more information about the `threadprivate` directive) variables in *list*, copy the master thread's value to the thread's private copy
- `reduction(`*operator* `:` *list*`)`
  Each of the variables in *list* will be marked for reduction by *operator*: Each thread will work in its own private copy of the variables, and the results will be combined at the end by *operator*.

3.3.2 `For` Construct

The *`for` construct* is used to divide iterations of a for loop among the threads that are currently running. There is an implied *barrier* (see section 3.3.6 for information about synchronization constructs) at then end of each for construct, unless removed by specifying the *nowait* clause.

The for construct consists of a for directive followed by a for loop in canonical form:

```
#pragma omp for [clauses]
for (var = lower_bound; var < upper_bound; var += increment)
   statement;
```

(Other variations of the for loop header are also allowed, see the OpenMP specification, Section 2.4.1, page 11, for details.)

The for construct accepts these clauses:

- `private(`*list*`), firstprivate(`*list*`), lastprivate(`*list*`)`
  Identical to those available for the parallel construct. *Lastprivate* functions analogously to *firstprivate*: after the for loop has been executed, the value generated by the thread which ran the last iteration of the loop is copied out to the variable outside the for loop
- `reduction(`*operator* `:` *list*`)`
  as for the parallel construct
- `ordered`
  makes it possible to use the 'ordered' construct within this for loop, see below
- `schedule(`*kind* `[,` *chunk_size*`])`
  selects the kind of scheduler and granularity of scheduling to be used
- `nowait`
  removes the implied *barrier* at the end of the for construct, allowing the participating threads to continue working without having to wait for slow threads

3.3.3 `Sections` Construct

The *`sections` construct* is used to let each thread in the currently active team perform a different task, all at once. If there are more sections in a sections construct than there are processors in the team executing it, each processor may be called upon to execute more than one of the sections. There is an implied *barrier* (see section 3.3.6 for information about synchronization constructs) at then end of each sections construct, unless removed by specifying the *nowait* clause.

The sections construct takes the form of a sections directive followed by a compound-statement, containing a series of sections, as below:

```
#pragma omp sections [clauses]
{
   [#pragma omp section]
      structured-block
   [#pragma omp section
      structured-block
   .
   .
   .]
}
```

The sections construct accepts these clauses, which function identically on the sections construct as on the for construct:

- `private(`*list*`), firstprivate(`*list*`), lastprivate(`*list*`)`
- `reduction(`*operator* `:` *list*`)`
- `nowait`

3.3.4 `Single` construct

The `single` *construct* is used to specify that, even within a parallel region, a certain piece of code is only to be executed by *one* of the threads (though not necessarily the master thread). As with the for anc sections constructs, there is an implied barrier at the end of each single construct, which can be removed using the *nowait* clause.

The single construct takes the following form:

```
#pragma omp single [clauses]
    structured-block
```

The single construct accepts these clauses, which function identically on the sections construct as on the for and sections constructs:

- `private(`*list*`)`, `firstprivate(`*list*`)`
- `nowait`

3.3.5 Combined Parallel Work-Sharing Constructs

The combined parallel work-sharing constructs combine the parallel construct with a work-sharing construct.

`Parallel For` Construct

```
#pragma omp parallel for [clauses]
for(var = lower_bound; var < upper_bound; var +=incr)
    statement
```

`Parallel Sections` Construct

```
#pragma omp parallel sections [clauses]
{
   [#pragma omp section]
       compoud-statement
   [#pragma omp section
       compound-statement
   .
   .
   .]
}
```

Each accepts the same clauses that either of the parallel or the work-sharing construct in question would accept, with the exception of `nowait`, since the implied barrier in the parallel construct can not be removed.

No, there is no `parallel single` *construct* – starting parallel execution just to let only one of the threads in the team execute the code inside it is not a useful exercise.

3.3.6 Master and Synchronization Constructs

OpenMP also defines a number of Constructs for synchronization purposes:

`Master` Construct

```
#pragma omp master
   compound-statement
```

Only the master thread of the current team will execute the compound-statement.

`Critical` Construct

```
#pragma omp critical [(name)]
  compound-statement
```

Each thread waits at the entry to a critical construct, until no other thread is executing a critical region with the same name. All unnamed critical regions map to the same unspecified name.

`Barrier` Directive

```
#pragma omp barrier
```

Specifies a point which *all* threads in the current team must reach before any one of them can proceed. Note, this is *not* a statement.

`Atomic` Construct

```
#pragma omp atomic
var op= expression
```

This ascertains that the assignment is performed atomically, that is, by only one thread at a time.

`Flush` directive

```
#pragma omp flush [(list)]
```

This synchronizes the calling thread's view of the shared variables in *list* with shared memory. If no list is specified, flushes all currently visible shared variables. After two threads have each flushed the same variable (and before either of them attempts to change it again), both threads have the same view of that variable.

`Ordered` construct

```
#pragma omp ordered
    compound-statement
```

This can only be used within a for construct which has the ordered clause speficied upon it. In that case, it ensures that the *compound-statement* will be executed in exactly the same order as it would, were the loop executed sequentially.

3.3.7 `Threadprivate` Directive

The final directive specified by OpenMP is the *`threadprivate` directive*:

```
#pragma omp threadprivate (list)
```

The threadprivate directive is used lexically outside of any function definition, to mark file-scope visible variables as being private to each thread. All references to threadprivate-marked variables go to the copy of that variable which belongs to the current thread.

3.4 A Small Example

After this highly theoretical introduction, an example – albeit a small one – will hopefully help clear up any confusion. On the left in Figure 4, we have a simple program, written for traditional, sequential execution, with comments to indicate where parallelization might be possible. On the right in the same figure, the same program has been annotated with OpenMP directives, which take the form of compiler pragmas. As you can see, the actual code is unchanged; a compiler could simply ignore the OpenMP pragmas, in which case the results – from compiling the sequential program on the left, and from compiling the OpenMP-annotated program on the right – would be identical. This is quite intentional, of course.

```
#define N 10000
typedef int array_t [N];
extern void foo();
extern void bar();

int main(int argc, char **argv) {
      int i;
      large_array_t v;

      /* this could be parallelized */     Ⓐ
      for (i = 0; i < N; i++) {            Ⓑ
            v[i] = 0;
      }

      printf("initialized array\n");       Ⓒ

      /* these could run in parallel */    Ⓓ
      foo();                               Ⓔ
      bar();                               Ⓕ
}
```

```
#define N 10000
typedef int array_t [N];
extern void foo();
extern void bar();

int main(int argc, char **argv) {
      int i;
      large_array_t v;

      #pragma omp parallel                       Ⓐ
      {
            #pragma omp for                      Ⓑ
            for (i = 0; i < N; i++) {
                  v[i] = 0;
            }
      }

      printf("initialized array\n");             Ⓒ

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

Figure 4, A Simple OpenMP Example

So, how did we get from the sequential program on the left, to the OpenMP program on the right? Here's a quick run-down:

First, we note that the `for` loop at Ⓑcan be parallelized. So, we take the code that we want to run in parallel, and place it inside a 'parallel construct' (at Ⓐ), OpenMP's construct for parallelizing execution. A parallel construct consists of the '`#pragma omp parallel`' line, and a compound statement, that is, a normal series of statements enclosed by a pair of curly braces ('`{`' and '`}`'), as usual. Conceptually, a parallel construct starts a *team* of threads at its beginning, lets every thread in the team execute the code inside the compound statement – this is called the 'parallel region' – then waits until all threads are finished, and puts them to rest.

Now we have a number of threads, each to run the code inside the parallel region – in this case, the `for` loop. But we don't want each thread to execute all iterations of the loop, we want each thread to grab a 'chunk' of the loop and just run that. Well, OpenMP has a construct for this, not surprisingly called the 'for construct'. We place one of these at Ⓑ. The for construct consists of the '`#pragma omp for`' line, followed by a simple for loop – basically, one where an index variable loops through a range of numbers. The OpenMP compiler will now assign each of the threads in the currently active team an appropriately sized chunk of the total range of the loop, so that each thread only executes part of the loop, but that all threads together will execute the whole loop. This ends both the for construct at Ⓑ, and the parallel region defined by Ⓐ.

At Ⓒ, we have a `printf` statement – this obviously shouldn't be done more than once, so we'll just keep it like it is.

Now, we happen to know that `foo()` and `bar()` are functions which do totally unrelated work, and that there are no interdependencies between them – they could be run in any order to get the correct result. In fact, we can even run them in parallel – so, let's do that. Parallel execution is commenced by placing the code in question inside a parallel region; this is done by wrapping the code inside a '`#pragma omp parallel { … }`' construct at 🅓, like before. As you can see, it is quite possible to have several different parallel regions in your program.

Now, what we want to do is to let `foo()` and `bar()` run in parallel; but they are not just different iterations of the same for loop as we had in 🅑, but instead we want to run to completely different code snippets, in parallel. Well, OpenMP has a construct for this also, the 'sections construct'. This explicitly marks up what parts of the code can be run in parallel, by grouping the code into 'sections', which can then be run in parallel, as you can see in 🅔and 🅕.

The for and sections constructs are examples of what OpenMP calls 'work sharing constructs' – they take some amount of work, and share it between threads. We have already seen that program execution after a parallel construct does not proceed until all threads in the team have finished executing the code in the parallel region – there is, in OpenMP parlance, an 'implied barrier' at the end of each parallel construct. There is also, by default, an implicit barrier at the end of each work sharing construct. This means that, unless you specifically request otherwise, you can rely on all iterations of a parallelized for loop to have been run, in any code that might follow the for construct.

A note to the prospective OpenMP programmer who prefers to place his opening brace at the end of the line: You can't do that here; all '`#pragma omp …`' constructs *explicitly* include an end-of-line, so you *have* to put the opening brace for the ensuing compound statement on the next line.

Now, you may quite correctly observe that it is a bit cumbersome, if you only want to run a for loop in parallel, to have to wrap that inside two constructs – the parallel construct which starts parallel execution, and the for construct which divides out the iterations between the threads. The designers of OpenMP apparently agree with you here, because they have defined 'combined parallel work-sharing constructs' which, as the name suggests, combine a parallel construct with a work-sharing construct. So, in order to parallelize a for loop, rather than having

```
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i < N; i++) {
      foo();
   }
}
```

you would have

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   foo();
}
```

which is much less crowding.

4 OdinMP/CCp

4.1 What is it?

OdinMP is a project at the Department of Information Technology at Lund Institute of Technology, with the goal of producing a set of freely available, portable implementations of the OpenMP standard for a variety of platforms. OdinMP/CCp is one of these implementations, implementing the OpenMP specification for the ANSI C programming language [2], producing code in C using the POSIX thread library, generally known as ‘pthreads’ [3], as the underlying threading mechanism. The suffix ‘CCp’ is an acronym of ‘from *C* to *C* with *p*threads’.

4.2 Why OdinMP/CCp?

OdinMP/CCp has the big advantage of being quite portable and platform-independent. All it requires of its target platform is that the POSIX threads library be available, which these days is more the rule than the exception for Unix-derived systems, as well as becoming more and more commonplace in other computing environments. Thus, a program written with OpenMP can be compiled and used on such systems, using OdinMP/CCp, even if no platform-specific OpenMP implementation is available.

Further, we regard OpenMP to be a good development in multiprocessing, one that should be spurred on and made available to programmers and researchers as widely as possible. OdinMP/CCp is a vehicle to ‘spread the word’ about OpenMP among people with different platforms and budgets, professionals and hobbyists alike.

OdinMP/CCp can also be used in development of a platform-specific OpenMP compiler, both for comparisons regarding execution, and as a prototype, since it is available with full source code.

4.3 Why might you use OdinMP/CCp?

One very simple example of this would be a home-built multiple-Pentium® PC running Linux™. No OpenMP-capable C compiler is available for this platform, but Linux has good support for POSIX threads. Using OdinMP/CCp, (some of) the power of this low-cost shared-memory multiprocessor can be harnessed.

Likewise, an experimental multiprocessing system or one in development may have good use for an existing OpenMP implementation for evaluation or development purposes.

OdinMP/CCp can also be used to perform compiler-neutral performance comparisons between different platforms.

5 Parallelization Overview

In this section I will give an overview of the basic principles employed by OdinMP/CCp in translating the OpenMP constructs and directives into C code with POSIX thread library calls. This is intended to introduce some of the major concepts, rather than provide a detailed look (I will provide that later in this paper). Much is omitted for clarity, and any names used here are illustrations only.

5.1 `Parallel` construct

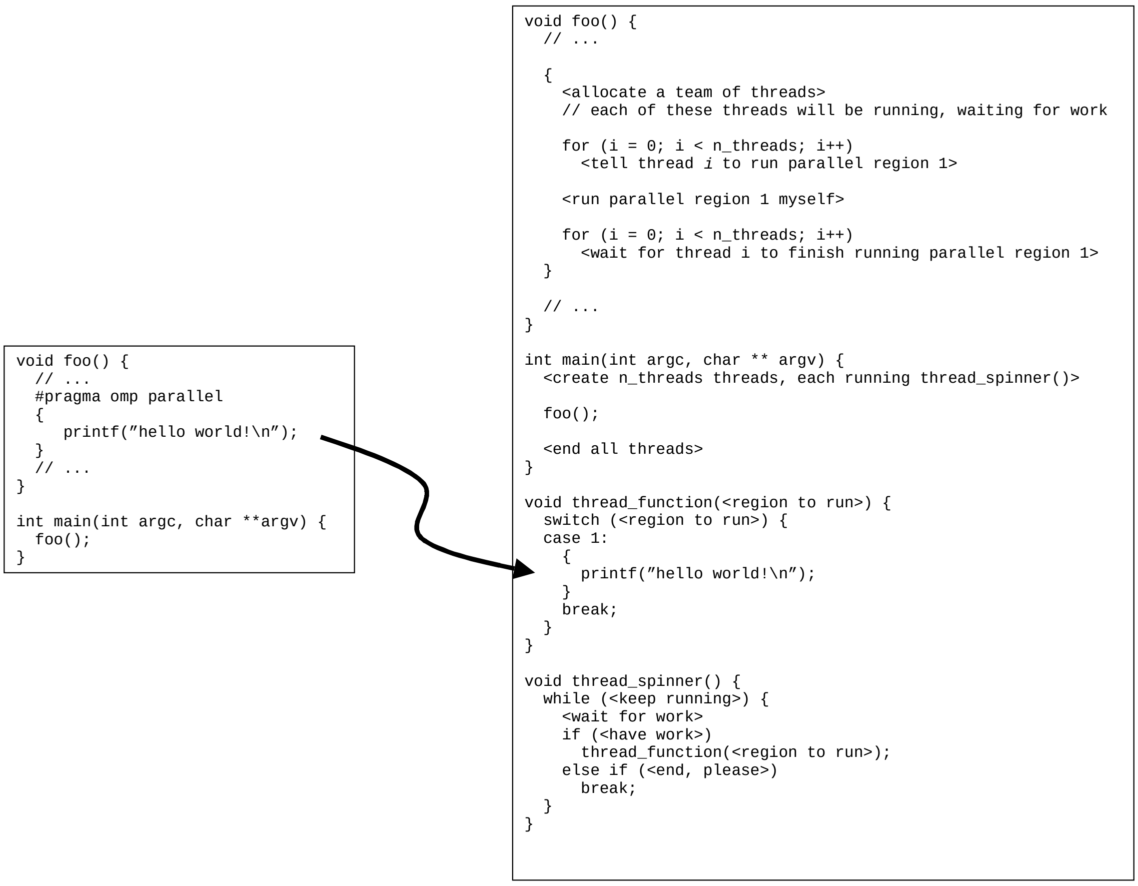
In the POSIX threads API, starting a thread requires designating a function for the new thread to run. So, anything that we want to be able to run in parallel must be inside, or called from, the function that each thread in the system runs. OdinMP/CCp’s basic paradigm for parallelization works as follows:

1. OdinMP/CCp defines a function `thread_spinner`, which waits (on a condition lock) for work, and calls `thread_function` to perform the actual task. Unless of course `thread_spinner` is told to finish spinning, in which case it returns, ending that thread’s life.
2. OdinMP also defines a function `thread_function`, which is basically a large, initially empty, `switch` statement.
3. 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 will be selected by its associated number. The parallel construct is replaced with code which

- allocates a team of threads
- tells each of the threads in the team to execute the parallel region in question
- runs the parallel region itself (as the master thread of the team)
- waits for all the other threads in the team to finish.

This is illustrated in Figure 5.



```
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;
  }
}
```

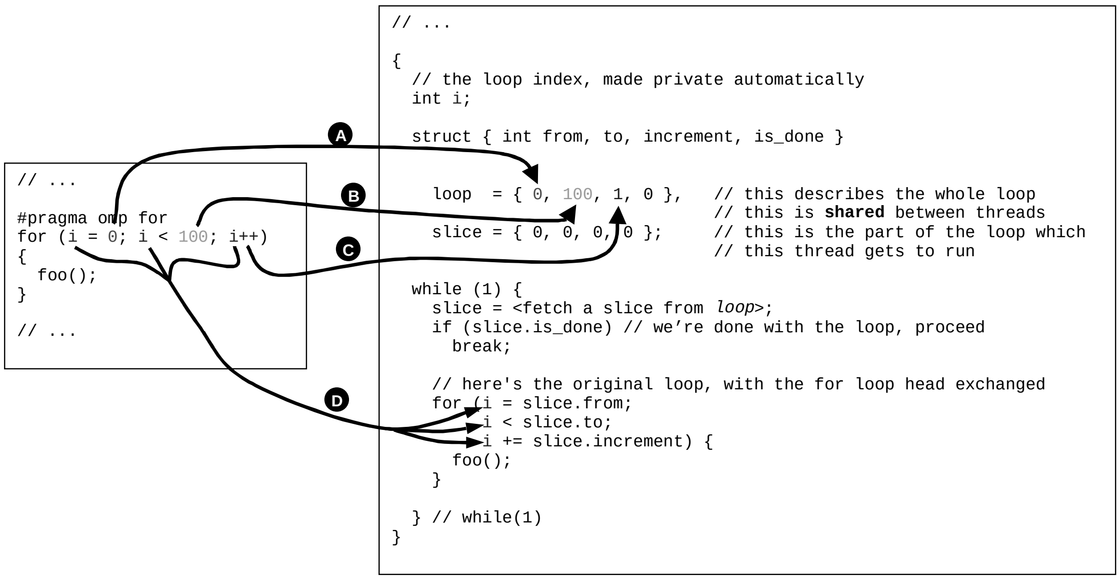
Figure 5, Parallel Construct

5.2 For construct

The `for` construct divides the iterations of a for loop into smaller slices, hands them to the different threads, which run the slices in parallel. To do this, we first extract the range, increment etc. of the whole loop from the for loop header. Then each thread loops around:

- Fetch a slice of the shared loop
- If there's no slice for us, then we are done and exit the loop
- Otherwise, iterate over the space of the slice

This is illustrated in Figure 6. The arrows indicate how OdinMP/CCp extracts information from the for loop header to find the loop index variable (Ⓐ), the loop initialization value (Ⓑ), the boundary value (Ⓒ) and the increment (Ⓓ).



```
// ...

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

// ...
```

```
// ...

{
  // 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();
    }

  } // while(1)
}
```

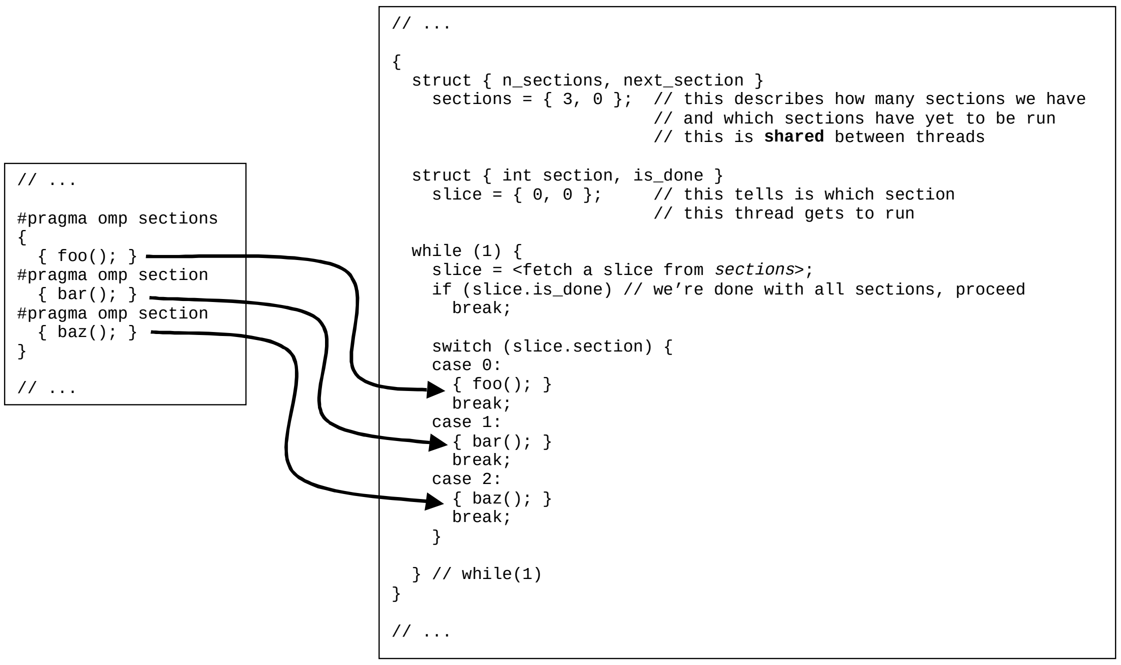
Figure 6, For Construct

5.3 `Sections` construct

The `sections` construct hands each section to a different thread, until each section has been executed. We achieve this much like we did for the `for` construct. We gather information about how many sections there are to be run, and give each section a unique number within this `sections` construct. Each thread then loops around:

- Fetch a slice of the sections – i.e., fetch the number of a section to run
- If there's no slice for us, then we are done and exit the loop
- Otherwise, execute the appropriate section.

Again, this is illustrated in Figure 7:



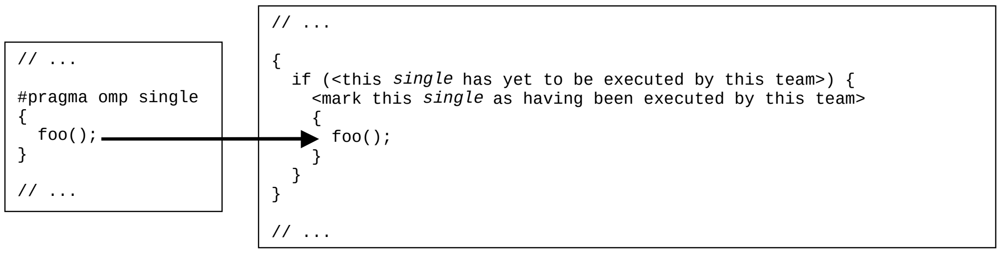
```
// ...

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

// ...
```

```
// ...

{
  struct { n_sections, next_section }
    sections = { 3, 0 };  // this describes how many sections we have
                          // and which sections have yet to be run
                          // this is shared between threads

  struct { int section, is_done }
    slice = { 0, 0 };     // this tells is which section
                          // this thread gets to run

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

    switch (slice.section) {
    case 0:
      { foo(); }
      break;
    case 1:
      { bar(); }
      break;
    case 2:
      { baz(); }
      break;
    }

  } // while(1)
}

// ...
```

Figure 7, Sections Construct

5.4 `Single` construct

The `single` construct works similarly to its cousins, except that only the first thread that attempts to execute it, actually does. Hence the 'slice' that each thread gets is either permission to execute the code in the `single` region, or not; no need to loop. Figure 8 should illustrate it:



```
// ...

#pragma omp single
{
  foo();
}

// ...
```

```
// ...

{
  if (<this single has yet to be executed by this team>) {
    <mark this single as having been executed by this team>
    {
      foo();
    }
  }
}

// ...
```

Figure 8, Single Construct

5.5 `Parallel work-sharing` constructs

A parallel work-sharing construct is handled as you might expect: a `parallel` construct immediately wrapped around the corresponding work-sharing (for or sections) construct.

5.6 `Threadprivate` directive

The threadprivate directive marks an otherwise global variable as being private to each thread. This means that for each thread other than the main thread in the program, a copy of each threadprivate variable must be made, and each access to a variable that is marked as threadprivate must go to the current thread's copy rather than the master thread's one.

OdinMP achieves this by allocating copies of the threadprivate variables in each thread other than the main thread. Each thread *including* the main thread also has a set of pointers to its threadprivate variables. Each reference to a threadprivate variable is then replaced by an expression which dereferences the current thread's pointer to the threadprivate variable in question.

6 A More Detailed Look at OdinMP Parallelization

I will present the process of parallelization and handling of OpenMP constructs, directives and clauses in detail, with examples as generated by the actual OdinMP compiler. The list of OdinMP types and symbols, as found in Appendix A, is very helpful for understanding this chapter, as is the OpenMP specification.

6.1 POSIX Threads, a Quick Introduction

The POSIX[*] threads API offers a variety of features for controlling multi-threaded execution of a program. OdinMP/CCp fortunately only uses a few of those, which I will briefly describe here. Full documentation for pthreads is available from <http://www.opengroup.org/unix/>.

6.1.1 Thread Creation and Termination

Pthreads defines a data type, `pthread_t`, which uniquely identifies a thread of execution. This is an opaque data type, in that the programmer who uses pthreads need never concern himself with the contents of any variable of type `pthread_t`.

To create a new thread, pthreads defines a routine with the following prototype:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

This declares `pthread_create` to be a function returning an integer, which takes as its arguments

1. a pointer to a `pthread_t` variable, *thread*
2. a pointer to a `pthread_attr_t` variable, *attr*
3. a pointer to a function taking a pointer to void as an argument and returning a pointer to void, *start_routine*
4. a pointer to void, *arg*

`pthread_create` starts a new thread with attributes according to the pthread attribute set *attr* points to, and stores the thread id at *thread*. The new thread will execute the routine *start_routine*, which will be called with *arg* as its argument. The newly created thread will terminate when it reaches the end of *start_routine*.

[*] The pthreads API was first defined in the IEEE POSIX 1003.1c standard, and is now a part of 'The Single UNIX Specification' from the Open Group (at <http://www.opengroup.org/>), which is what I refer to in this document.

6.1.2 Synchronization

Pthreads offers two facilities for synchronization between threads: mutual-exclusion locks (called *mutexes*), and *conditions*.

A mutex is defined by a variable of type `pthread_mutex_t`. It can be locked by a call to `int pthread_mutex_lock(pthread_mutex_t *mutex)`, and unlocked by calling `int pthread_mutex_unlock(pthread_mutex_t *mutex)`. While a mutex is locked by one thread, no other thread can lock it. A call to pthread_mutex_lock will not return until the mutex has been successfully locked; thus, the calling thread will wait until no other thread has the mutex in question locked. This ensures that only one thread at a time can execute whatever code is bracketed by locking and unlocking a mutex.

A condition carries the type `pthread_cond_t`. Conditions are used in conjunction with mutexes, to wait for a certain event or signal that a certain even has occurred.

A thread that wishes to wait for a condition to occur calls `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`. This places the calling thread in a queue of threads which wait for `cond` to occur. While the thread is waiting, `mutex` is unlocked; after `pthread_cond_wait` returns, `mutex` has been re-locked.

To signal that a condition has occurred, a thread calls `int pthread_cond_signal(pthread_cond_t *cond)` or `int pthread_cond_broadcast(pthread_cond_t *cond)`. The difference between them is that `pthread_cond_signal` only informs one of the threads that are waiting for `cond`, whereas `pthread_cond_broadcast` tells all threads.

Let me give an example to make things clearer:

```
/*variables */
int x, y;
pthread_mutex_t mut;
pthread_cond_t cond;


/* Thread 1: */
/* wait until x > y */
pthread_mutex_lock(&mut);                                Ⓐ
while (!x > y) {                                         Ⓑ
   pthread_cond_wait(&cond, &mut);                       Ⓒ
}
/* operate on x and y */                                 Ⓓ
pthread_mutex_unlock(&mut);                              Ⓔ


/* Thread 2: */                                          Ⓕ
pthread_mutex_lock(&mut);                                Ⓖ
/* modify x and y */                                     Ⓗ
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);                              Ⓘ
```

The variables *x* and *y* are protected by a mutex, *mut* – any thread that wants to work on *x* or *y* has to lock *mut* first. There is also a condition *cond*, which is used to signal that *x* is greater than *y*.

Thread 1 wishes to work on *x* and *y*, so it locks *mut* (Ⓐ). However, it requires *x* to be greater than *y* to be able to do its work; so it lays itself in a loop (Ⓑ) which, while *x* is not greater than *y*, waits (Ⓒ) for some other thread to signal that *cond* has occurred. Once *cond* has occurred and *x* is indeed greater than *y*, thread 1 can proceed to operate on *x* and *y* (🅓) and finally unlock *mut* (🅔).

Thread 2 also works with *x* and *y*, so it locks *mut* (🅕). After it has finished modifying them (🅖), it checks whether *x* is greater than *y* and if so, tells any waiting threads that *cond* has occurred (🅗). Finally, it unlocks *mut* (🅘).

6.2 Project File Structure

A program is built from a set of input source files, written in C. OdinMP parses these, and generates its own set of C language source files as a result. A simple program will go through the following transition:

Original project	OdinMP project
Files in Project:	Files in Project:
• foo.c	• foo.c
• foo.h	• main.c
• main.c	• odinmp.c

Since all preprocessing will be done by OdinMP/CCp, header files ‘disappear’ from the project; but an additional source file – ‘odinmp.c’ – is generated.

6.3 Common Source File Changes

Each source code file will go through the following changes:

```
void foo()
{
    /* … */
}

int main (int argc, char **argv)
{
    /* … */
}
```

```
/* Type Definitions and External Declarations for OdinMP use                A
   omitted for brevity */

void foo()
{
    /* … */
}

int odinmp_original_main(int argc, char **argv)                             B1
{
   /* … */
}

/* Generated extra code: */                                                 C
void odinmp_thread_samp_c(odinmp_thread_data_t * odinmp_td)
{
    switch (((odinmp_thread_info_t *) (odinmp_td))->run_region) {           D
    }
}

void odinmp_thread_init_samp_c(odinmp_thread_data_t * odinmp_td, int i)     E
{
    if (((odinmp_thread_info_t *) odinmp_td)->id == 0) {
        odinmp_thread_init(odinmp_td, i + 1);                               F
    } else {
        odinmp_thread_init(odinmp_td, i + 1);                               G
    }

}

int main (int argc, char **argv)                                            B2
{
    odinmp_parse_env();
    odinmp_parse_args(&argc, argv);
    odinmp_init();
    atexit(odinmp_atexit);
    return odinmp_original_main(argc, argv);
}
```

Figure 9, Common Source File Changes

- Such type defnitions and external symbol declarations as OdinMP/CCp needs, are added before the file’s source code (at bullet 🅐).

- If this file contains a function called `main`, this is renamed to `odinmp_original_main` (B1), and a new `main` function is defined (B2), as you can see in the figure, which sets up the OdinMP/CCp environment before running `odinmp_original_main`
- Two 'standard' functions are defined:
  - `odinmp_thread_`*`filename_c`*`()` (at C)
  - `odinmp_thread_init_`*`filename_c`*`()` (at E)

  where *filename_c* represents the file's name with all non-alphanumeric characters replaced with underscores, thus creating function names which are unique to each file We will return to these functions later.

6.4 *`Parallel`* *Construct*

When OdinMP/CCp encounters a parallel region such as

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

the following occurs:

- The parallel region is assigned a unique identifying number; that is, unique to the whole project, not just to the current file. (In the example below, this identifying number is zero, '0', as it will always be for the first / only parallel region in a project.)
- If the parallel construct contains any firstprivate variables, or shared variables which are not globally visible, then OdinMP/CCp declares a structure `odinmp_par_0_shm_t` to hold a pointer to each of those variables. In our example it will look like this:
  ```
  typedef struct {
     int (*s);                        /* shared variable */
     int (*fp);                       /* firstprivate variable */
  } odinmp_par_0_shm_t;
  ```
- If there are any reduction variables in the parallel construct, then OdinMP/CCp declares a structure `odinmp_par_0_prm_t` to hold a working copy of each reduction variable. In our example it will look like this:
  ```
  typedef struct {
     int rdx;
  } odinmp_par_0_prm_t;
  ```
- The entire parallel region is replaced by the following code:

```
void foo ()
{
   int s, p, fp, rdx;
   /* … */

   /*  # pragma omp  */
   {
      /* we need a loop counter and an id */
      int odinmp_i, odinmp_id;
```

```
/* for allocating a team of threads */
odinmp_team_t *odinmp_team;
odinmp_team_list_t *odinmp_tl;
odinmp_thread_data_t *odinmp_td, *odinmp_master_td;

/* initialize the pointers to the shared variables */
odinmp_shared_data.par_0.s = &s;                                          A

/* initialize pointers to firstprivate variables, if any */
odinmp_shared_data.par_0.fp = &fp;                                        B

/* allocate a team of threads */
/* ok, let’s allocate as many as we can get */                            C
odinmp_team = odinmp_allocate_team (0);
odinmp_master_td = odinmp_thread_datas[odinmp_team->ids[0]];
/* for each thread, initialize the private data, and dispatch
   the thread  */
for (odinmp_i = 0; odinmp_i < odinmp_team->n; odinmp_i++) {
  odinmp_id = odinmp_team->ids[odinmp_i];
  odinmp_td = odinmp_thread_datas[odinmp_id];

  ((odinmp_thread_info_t *) (odinmp_td))->run_region = 0;                D
  if (((odinmp_thread_info_t *) (odinmp_td))->team) {
     odinmp_tl =
        (odinmp_team_list_t *) malloc (sizeof (odinmp_team_list_t));
     odinmp_tl->next = ((odinmp_thread_info_t *) (odinmp_td))->teams;
     ((odinmp_thread_info_t *) (odinmp_td))->teams = odinmp_tl;
  }
  ((odinmp_thread_info_t *) (odinmp_td))->team = odinmp_team;

  /* copy data from master thread into those threadprivate
     variables that are listed in the copyin clause, if any */           E

  /* initialize reduction variables */
  odinmp_td->par_0.rdx = 0;                                               F

  odinmp_dispatch (odinmp_td);                                            G
}

odinmp_thread (odinmp_thread_datas[odinmp_team->ids[0]]);                 H


/* wait for all other threads to finish running this region */
for (odinmp_i = 0; odinmp_i < odinmp_team->n; odinmp_i++) {
  odinmp_id = odinmp_team->ids[odinmp_i];
  odinmp_td = odinmp_thread_datas[odinmp_id];
  if (odinmp_i != 0)                                                      I
     odinmp_wait_finished (odinmp_id);


  ((odinmp_thread_info_t *) (odinmp_td))->run_region = -1;               J
  if (((odinmp_thread_info_t *) (odinmp_td))->teams) {
     odinmp_tl = ((odinmp_thread_info_t *) (odinmp_td))->teams;
     ((odinmp_thread_info_t *) (odinmp_td))->teams = odinmp_tl->next;
     ((odinmp_thread_info_t *) (odinmp_td))->team = odinmp_tl->team;
     free (odinmp_tl);
  } else
     ((odinmp_thread_info_t *) (odinmp_td))->team = ((void *) 0);

  /* reduce the reduction variables */
  rdx = rdx + odinmp_td->par_0.rdx;                                       K
}
```

```
      /* deallocate team of threads */
      odinmp_free_team (odinmp_team);                                🅛
   }

   /* … */
}
```

The above code does the following: At 🅐, OdinMP/CCp initializes the pointers in the shared memory structure, to point to those shared variables which are not globally visible (and thus not directly accessible from `odinmp_thread_samp_c()`). At 🅑, OdinMP/CCp similary initalizes the pointers to the ‘originals’ of the firstprivate variables. At 🅒, OdinMP/CCp allocates a team of threads, as many as it can get hold of, and marks the current thread as the master thread of that team.
At this point, all threads in the team (other than the current thread) are waiting in `odinmp_thread_spinner()` at BULLET.
Then, for each thread in the newly allocated team, OdinMP/CCp tells it to run the correct parallel region (at 🅓) and tells it what team it is a member of. At 🅔, it copies in the master thread’s values of any threadprivate variables named in the copying clause to this parallel construct (in this example, there aren’t any). At 🅕, the reduction variables (private to each thread) are initialized, and finally at 🅖, the freshly initialized thread is dispatched to do its assigned work, by setting the condition that the thread in question has work to do. The affected thread will then call `odinmp_thread()`, which will in turn call `odinmp_thread_samp_c()`.
Now, the current thread itself executes the pertinent parallel region as the master thread of the team, by calling `odinmp_thread()` at 🅗.
After returning from that, OdinMP/CCp will attempt to ‘collect’ all threads in the team: It waits for the thread to finish (🅘), resets that thread’s team information (🅙) and finally reduces that thread’s working copy of each reduction variable into the master copy (🅚). Finally, after all threads have been collected, the team is empty and thus can be deallocated – at 🅛.

- Further, in the generated routine `odinmp_thread_samp_c()` (in Figure 9, Common Source File Changes, at 🅐), a case is added, selected by the identifying number chosen above (0 in this example), as follows:

```
void odinmp_thread_samp_c (odinmp_thread_data_t * odinmp_td)
{
   switch (((odinmp_thread_info_t *) (odinmp_td))->run_region) {
   case 0:                                                           🅐
      /* outer scope: variables */
      {
        /* declare & define variables here */
        int fp = (*odinmp_shared_data.par_0.fp);                     🅑
        int p;                                                       🅒

        /* original code of parallel region here */
        {
           bar ();
        }                                                            🅓
      }
      break;
   }
}
```

As mentioned above, `odinmp_thread_samp_c()` will be called from `odinmp_thread_spinner()` when a parallel region defined in samp.c is to be executed by that thread. This code will execute the parallel region as follows: First, the relevant part of the switch statement is selected by the case label with the same number as we have assigned to the corresponding parallel region (Ⓐ). OdinMP/CCp then opens a new stack frame, and on that frame declares and thus allocates each private variable for the parallel region (Ⓑ and Ⓒ). Those that are also firstprivate get initialized (at Ⓑ) through the corresponding pointer that was set before the current thread was dispatched. And then OdinMP/CCp just includes the code of the parallel region for the current thread to run, at Ⓓ.

6.5 For Construct

A for construct like

```
void foo() {
  int s;                      /* shared */
  int a = 0, b = 100, c = 3; /* loop lower bound, upper bound, and increment */
  int i, j;                   /* loop counters */
  int fp = 0, flp = 0, lp;    /* FirstPrivate, First&LastPrivate, LastPrivate */
  int rdx = 0;                /* reduction variable */

  /* ... */
#pragma omp parallel
  {
    #pragma omp for \
      private(j) firstprivate(fp, flp) lastprivate(flp, lp) \
      reduction(+ : rdx) ordered
    for (i = a; i < b; i += c)
      rdx += bar(i, j, s, fp, &flp, &lp);
  }
  /* ... */
}
```

generates the following code:

```
/*  # pragma omp  */
{
   /* private variables;
      if a variable is also firstprivate, then initialize it
      from the corresponding outer variable
      if a variable is a reduction variable, then initialize it
      corresponding to the reduction operator */
   int odinmp_flp_fp = (*odinmp_shared_data.par_0.fp);                  Ⓐ
   int odinmp_flp_flp = (*odinmp_shared_data.par_0.flp);
   int odinmp_flp_lp;
   int odinmp_rdx_rdx = 0;
   int j;
   int i;

   /* odinmp variables */
   /* wsc info etc */
   odinmp_for_slice_t odinmp_slice = {0L, 0L, 0, 0, 0, 0};              Ⓑ
   odinmp_wsc_info_t odinmp_wsc_info;
   int odinmp_in_order = 0;

   /* for slice fetching */
   odinmp_team_t *odinmp_team = ((odinmp_thread_info_t *) (odinmp_td))->team; Ⓒ
   pthread_mutex_t *odinmp_team_lock = &odinmp_team->lock;
   /* record info about this work sharing construct */
   odinmp_wsc_info.next = ((odinmp_thread_info_t *) (odinmp_td))->wsc_info;    D
   ((odinmp_thread_info_t *) (odinmp_td))->wsc_info = &odinmp_wsc_info;
   odinmp_wsc_info.type = odinmp_wsc_type_for;
   odinmp_wsc_info.u.wsc_for.slice = &odinmp_slice;

   pthread_mutex_lock (odinmp_team_lock);                                      E

   /* find the lap list, if any */
   for (odinmp_wsc_info.u.wsc_for.lap_list = odinmp_team->for_laps[0];         F
        odinmp_wsc_info.u.wsc_for.lap_list &&
odinmp_wsc_info.u.wsc_for.lap_list->lap > odinmp_td->for_0.odinmp_lap;
        odinmp_wsc_info.u.wsc_for.lap_list =
odinmp_wsc_info.u.wsc_for.lap_list->next);
                                                                               G
   if (!odinmp_wsc_info.u.wsc_for.lap_list ||
odinmp_wsc_info.u.wsc_for.lap_list->lap != odinmp_td->for_0.odinmp_lap) {
      ;

      odinmp_wsc_info.u.wsc_for.lap_list = (odinmp_for_lap_list_t *)           H
         malloc (sizeof (odinmp_for_lap_list_t));
      odinmp_wsc_info.u.wsc_for.lap_list->lap = odinmp_td->for_0.odinmp_lap;
      odinmp_wsc_info.u.wsc_for.lap_list->n_left_to_leave = odinmp_team->n;
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.lb =
(*odinmp_shared_data.par_0.a);
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.incr =
(*odinmp_shared_data.par_0.c);
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.b =
((*odinmp_shared_data.par_0.b));
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.scheduler.type =
odinmp_scheduler_static;
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.scheduler.chunk_size =
(odinmp_wsc_info.u.wsc_for.lap_list->for_data.b -
odinmp_wsc_info.u.wsc_for.lap_list->for_data.lb) / (double) odinmp_team->n;
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.chunk_size =
odinmp_wsc_info.u.wsc_for.lap_list->for_data.scheduler.chunk_size;

      odinmp_wsc_info.u.wsc_for.lap_list->next = odinmp_team->for_laps[0];
      odinmp_team->for_laps[0] = odinmp_wsc_info.u.wsc_for.lap_list;

      /* if an ordered clause is specified on for construct */                 I
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.in_order = 0;
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.next_in_order = 0;
      pthread_mutex_init (&odinmp_wsc_info.u.wsc_for.lap_list-
>for_data.order_lock, ((void *) 0));
      pthread_cond_init (&odinmp_wsc_info.u.wsc_for.lap_list-
>for_data.new_order, ((void *) 0));
   }
   pthread_mutex_unlock (odinmp_team_lock);                                    J

   while (1) {                                                                 K
      /* fetch a slice */

      /* reset slice */                                                        L
      odinmp_slice.last = 0;
      odinmp_slice.done = 0;
      odinmp_slice.lb = 0;
      odinmp_slice.incr = odinmp_wsc_info.u.wsc_for.lap_list->for_data.incr;
      odinmp_slice.b = 0;

      {                                                                        M
         int i = ((odinmp_thread_info_t *) (odinmp_td))->id
```

```
            + odinmp_slice.i * odinmp_team->n;

         odinmp_slice.lb =
            (long long int) (odinmp_wsc_info.u.wsc_for.lap_list->for_data.lb +
i * odinmp_wsc_info.u.wsc_for.lap_list->for_data.chunk_size);

         if (odinmp_slice.lb < odinmp_wsc_info.u.wsc_for.lap_list->for_data.b)
{
            odinmp_slice.b =
               (long long int) (odinmp_wsc_info.u.wsc_for.lap_list->for_data.lb
+ (i + 1) * odinmp_wsc_info.u.wsc_for.lap_list->for_data.chunk_size);
            if (!(odinmp_slice.b < odinmp_wsc_info.u.wsc_for.lap_list-
>for_data.b)) {
               odinmp_slice.b = odinmp_wsc_info.u.wsc_for.lap_list->for_data.b;
               odinmp_slice.last = 1;
            }
            odinmp_slice.in_order = i;

            odinmp_slice.i++;
         } else {
            odinmp_slice.done = 1;
         }
      }

      if (odinmp_slice.done) {                                                 N
         /* increment lap counter */
         odinmp_td->for_0.odinmp_lap++;
         /* ... and exit the while(1) loop */
         break;
      }
      /* here’s the original loop, with the for loop head exchanged
         and variable references replaced as well */                           O
      for (i = odinmp_slice.lb;
           i < odinmp_slice.b;
           i += odinmp_slice.incr)
         odinmp_rdx_rdx += bar (i, j, (*odinmp_shared_data.par_0.s),
odinmp_flp_fp, &odinmp_flp_flp, &odinmp_flp_lp);

      /* update order counter */
      pthread_mutex_lock (&odinmp_wsc_info.u.wsc_for.lap_list-                 P
>for_data.order_lock);
      odinmp_wsc_info.u.wsc_for.lap_list->for_data.in_order++;
      pthread_cond_broadcast (&odinmp_wsc_info.u.wsc_for.lap_list-
>for_data.new_order);
      pthread_mutex_unlock (&odinmp_wsc_info.u.wsc_for.lap_list-
>for_data.order_lock);
                                                                               Q
      /* did we run the last iteration ? */
      if (odinmp_slice.last) {
         /* yes, so copy data out from lastprivate variables */
         (*odinmp_shared_data.par_0.flp) = odinmp_flp_flp;
         (*odinmp_shared_data.par_0.lp) = odinmp_flp_lp;
      }                                                                        R
   }                           /* while(1) */

   /* restore work sharing construct info */
   ((odinmp_thread_info_t *) (odinmp_td))->wsc_info = odinmp_wsc_info.next;    S

   pthread_mutex_lock (odinmp_team_lock);                                      T

   /* reduce reduction variables */                                            U
   (*odinmp_shared_data.par_0.rdx) = (*odinmp_shared_data.par_0.rdx) +
odinmp_rdx_rdx;
                                                                               V
   /* update the lap list */
   if (--odinmp_wsc_info.u.wsc_for.lap_list->n_left_to_leave == 0) {
      odinmp_for_lap_list_t *oll = odinmp_team->for_laps[0];

      ;
      if (oll == odinmp_wsc_info.u.wsc_for.lap_list) {
         odinmp_team->for_laps[0] = oll->next;
      } else {
         while (oll->next != odinmp_wsc_info.u.wsc_for.lap_list)
            oll = oll->next;
         oll->next = odinmp_wsc_info.u.wsc_for.lap_list->next;
      }
      free (oll);
   }                                                                Ⓦ
   pthread_mutex_unlock (odinmp_team_lock);
   /* barrier at the end of parallel for
      remove this section if nowait is specified */                 Ⓧ
   odinmp_barrier (odinmp_team,
                   &odinmp_team->for_barriers[0]);
}
```

At Ⓐ, a private copy of each private variable is allocated. Those that are also marked as firstprivate are initialized from the corresponding shared variable. Also, a private working copy of each reduction variable is allocated and initialized according to its reduction operator. At Ⓑ, OdinMP/CCp declares a variable to hold the information for the current slice to be run, information about this work-sharing construct, and ordering information. OdinMP/CCp then declares some variables it uses as shortcuts to the current thread's team, and the team's mutual-exclusion lock, at Ⓒ. The work-sharing construct information is then initialized (Ⓓ).

Now, OdinMP/CCp locks the team's mutex (Ⓔ) – to ensure that only one thread in this team eecutes the following code at a time. It then searches for an existing 'lap descriptor' (Ⓕ).

'Laps' are used to keep track of the various time a given team of threads executes the same work sharing construct – for instance, a for construct might be situated inside another loop, or the for construct might be in a subroutine that is called several times from within the same parallel region. Conceivably, two threads in the same team might be executing different laps of the same work-sharing construct simultaneously, so the team needs to keep track of *all* the laps which currently have active threads on this work-sharing construct, hence it keeps lap descriptors in a linked list. Laps are numbered sequentially, starting with zero.

If OdinMP/CCp can find (Ⓖ) a lap descriptor matching the lap that the current threads wants to run, then the current thread is not the first to enter this lap; thus, this lap is already initialized and everything is fine. If, however, no matching lap descriptor can be found, then (Ⓗ) the current thread is the first in its team to enter this lap, so it allocates a new lap descriptor and initializes it (setting the lap number, the loop bounds and increment, and scheduling information). If the ordered clause has been specified on the current for construct, then the necessary information for that is initialized as well, at Ⓘ. Finally, the team's mutex is unlocked to let another team in the thread get at the lap descriptor (Ⓙ).

At Ⓚ, the current thread enters an 'infinite' loop which repeats the process of fetching,and then doing, work. First, the information about the current slice is reset (Ⓛ). Second, using the appropriate scheduler as specified in the 'schedule' clause (or the default scheduler, as in this example), the slice information for the current thread is fetched (Ⓜ). I will not go into detail on this process now.

If the slice information indicates that there is no more work with this thread to do (Ⓝ), then it exits from the 'infinite' loop and resumes execution at Ⓡ. Otherwise it stays inside the loop, proceeding to execute its slice of the for loop (🅞), using the slice information provided by the scheduler.

At 🅟, order information is updated if the ordered clause is specified on the current for construct. Then it is checked whether the just-executed slice contained the secuentially last iteration of the loop (🅠), in which case data is pushed out from any variables marked as lastprivate to their shared counterparts outside the for construct. The ‘infinite’ loop ends at 🅡.

After having exited from the ‘infinite’ loop, OdinMP/CCp restores the work-sharing construct information for the current thread (🅢), since it it now exiting from the current for construct. It then locks the team mutex again (🅣) and reduces its working copies of any reduction variables into the master copy, at 🅤. Likewise, it notes that it is exiting the current lap, and checks whether it might be the last thread in the team to do so (🅥); if it is, then this laps’ information is no longer needed and can be deallocated. All team-critical work being done, the team’s mutex is once more unlocked (🅦).

Finally, unless the ‘nowait’ clause is specified on the for construct, OdinMP/CCp waits at 🅧 for all other threads in the team, by means of a barrier.

6.6 Sections Construct

For the following code snippet

```
void foo ()
{
   int s;                     /* shared */
   int p;                     /* private */
   int fp = 0, flp = 0, lp;  /* FirstPrivate, First&LastPrivate, LastPrivate */
   int rdx = 0;               /* reduction variable */

   /* ... */
   #pragma omp parallel
   {
      #pragma omp sections \
      private(p) firstprivate(fp, flp) lastprivate(flp, lp) \
      reduction(+ : rdx)
      {
         {
            rdx += section1 (fp, &flp, &lp, &p);
         }
         #pragma omp section
         {
            rdx += section2 (fp, &flp, &lp, &p);
         }
         #pragma omp section
         {
            rdx += section3 (fp, &flp, &lp, &p);
         }
      }
   }
   /* ... */
}
```

OdinMP/CCp generates the following code:

```
/*  # pragma omp  */  {
   /* private variables;
      if a variable is also firstprivate, then initialize it
      from the corresponding outer variable
      if a variable is a reduction variable, then initialize it
      corresponding to the reduction operator */
   int odinmp_flp_fp = (*odinmp_shared_data.par_0.fp);                    A
   int odinmp_flp_flp = (*odinmp_shared_data.par_0.flp);
   int odinmp_flp_lp;
   int odinmp_rdx_rdx = 0;
   int p;

   /* odinmp variables */                                                 B
   /* wsc info etc */
   odinmp_sect_slice_t odinmp_slice;
   odinmp_wsc_info_t odinmp_wsc_info;

   /* for slice fetching */                                               C
   odinmp_team_t *odinmp_team = ((odinmp_thread_info_t *) (odinmp_td))->team;

   /* record info about this work sharing construct */                    D
   odinmp_wsc_info.next = ((odinmp_thread_info_t *) (odinmp_td))->wsc_info;
   ((odinmp_thread_info_t *) (odinmp_td))->wsc_info = &odinmp_wsc_info;
   odinmp_wsc_info.type = odinmp_wsc_type_sect;

   pthread_mutex_lock (&odinmp_team->lock);                               E

   for (odinmp_wsc_info.u.wsc_sect.lap_list = odinmp_team->sect_laps[0];  F
       odinmp_wsc_info.u.wsc_sect.lap_list &&
odinmp_wsc_info.u.wsc_sect.lap_list->lap > odinmp_td->sect_0.odinmp_lap;
       odinmp_wsc_info.u.wsc_sect.lap_list =
odinmp_wsc_info.u.wsc_sect.lap_list->next);

   if (!odinmp_wsc_info.u.wsc_sect.lap_list ||
odinmp_wsc_info.u.wsc_sect.lap_list->lap != odinmp_td->sect_0.odinmp_lap) {   G
      odinmp_wsc_info.u.wsc_sect.lap_list = (odinmp_sect_lap_list_t *)    H
        malloc (sizeof (odinmp_sect_lap_list_t));
      odinmp_wsc_info.u.wsc_sect.lap_list->lap = odinmp_td->sect_0.odinmp_lap;
      odinmp_wsc_info.u.wsc_sect.lap_list->n_left_to_leave = odinmp_team->n;
      odinmp_wsc_info.u.wsc_sect.lap_list->sect_data.n_left = 3;
      odinmp_wsc_info.u.wsc_sect.lap_list->next = odinmp_team->sect_laps[0];
      odinmp_team->sect_laps[0] = odinmp_wsc_info.u.wsc_sect.lap_list;
   }
   pthread_mutex_unlock (&odinmp_team->lock);                             I

   while (1) {                                                            J
      /* reset slice */
      odinmp_slice.last = 0;                                              K
      odinmp_slice.done = 0;

      pthread_mutex_lock (&odinmp_team->lock);                            L

      if (odinmp_wsc_info.u.wsc_sect.lap_list->sect_data.n_left > 0) {    M
        odinmp_slice.section = 3 - (odinmp_wsc_info.u.wsc_sect.lap_list-  N
>sect_data.n_left--);
        odinmp_slice.done = 0;
         /* are we performing the sequentially last section? */          O
         odinmp_slice.last = (odinmp_slice.section == 2);
      } else {                                                            P
        odinmp_slice.section = 0;
        odinmp_slice.done = 1;
        odinmp_slice.last = 0;
      }

      pthread_mutex_unlock (&odinmp_team->lock);                          Q

      if (odinmp_slice.done) {                                            R
        /* increment the lap counter */
        odinmp_td->sect_0.odinmp_lap++;
        /* ... and exit the while(1) loop */
        break;
      }
      switch (odinmp_slice.section) {                                          S
      case 0:                                                                  S1
        {
          odinmp_rdx_rdx += section1 (odinmp_flp_fp, &odinmp_flp_flp,
&odinmp_flp_lp, &p);
        }
        break;
      case 1:                                                                  S2
        {
          odinmp_rdx_rdx += section2 (odinmp_flp_fp, &odinmp_flp_flp,
&odinmp_flp_lp, &p);
        }
        break;
      case 2:                                                                  S3
        {
          odinmp_rdx_rdx += section3 (odinmp_flp_fp, &odinmp_flp_flp,
&odinmp_flp_lp, &p);
        }
        break;
      default:
        ;
      }

      /* did we just execute the last section? */                              T
      if (odinmp_slice.last) {
        /* yes we did, copy out the lastprivate variables */
        (*odinmp_shared_data.par_0.flp) = odinmp_flp_flp;
        (*odinmp_shared_data.par_0.lp) = odinmp_flp_lp;
      }
   }

   /* restore work sharing construct info */
   ((odinmp_thread_info_t *) (odinmp_td))->wsc_info = odinmp_wsc_info.next;   U

   pthread_mutex_lock (&odinmp_team->lock);                                    V

   /* reduce reduction variables */
   (*odinmp_shared_data.par_0.rdx) = (*odinmp_shared_data.par_0.rdx) +        W
odinmp_rdx_rdx;
   /* update the lap list */
   if (--odinmp_wsc_info.u.wsc_sect.lap_list->n_left_to_leave == 0) {          X
      odinmp_sect_lap_list_t *oll = odinmp_team->sect_laps[0];

      if (oll == odinmp_wsc_info.u.wsc_sect.lap_list) {
        odinmp_team->sect_laps[0] = oll->next;
      } else {
        while (oll->next != odinmp_wsc_info.u.wsc_sect.lap_list)
          oll = oll->next;
        oll->next = odinmp_wsc_info.u.wsc_sect.lap_list->next;
      }
      free (oll);
   }
   pthread_mutex_unlock (&odinmp_team->lock);                                  Y
   /* barrier at the end of sections
      remove this if nowait is specified */
   odinmp_barrier (odinmp_team,                                                Z
                &odinmp_team->sect_barriers[0]);
}
```

This code is quite similar to that generated for a for construct, not entirely surprisingly perhaps. At 🅐, a private copy of each private variable is allocated. Those that are also marked as firstprivate are initialized from the corresponding shared variable. Also, a private working copy of each reduction variable is allocated and initialized according to its reduction operator. At 🅑, OdinMP/CCp declares a variable to hold the information for the current slice to be run and information about this work-sharing construct. OdinMP/CCp then declares a variable it uses as a shortcut to the current thread's team at 🅒. The work-sharing construct information is then initialized (🅓).

Now, OdinMP/CCp locks the team's mutex (🅔) – to ensure that only one thread in this team executes the following code at a time. It then searches for an existing 'lap descriptor' (🅕), and if necessary (🅖) allocates and initializes one (🅗). Finally, the team's mutex is unlocked to let another team in the thread get at the lap descriptor (🅘).

At 🅙, the current thread enters an 'infinite' loop which repeats the process of fetching,and then doing, work. First, the information about the current slice is reset (🅚). Then the team's mutex is once more locked (🅛). If there is work left to be done by this team (🅜), then we fetch a slice for this thread (🅝) and check wether this also happens to be the last section in this sections construct (🅞), otherwise the slice is set to indicate that we are done here (🅟); now we can unlock the team's mutex (🅠). If the slice information indicates that there is no more work with this thread to do (🅡), then it exits from the 'infinite' loop and resumes execution at 🅤. Otherwise it stays inside the loop and proceeds to the switch statement at 🅢, which selects which of the sections S1, S2, S3 to execute according to the information in the slice we fetched earlier.

At 🅣, OdinMP/CCp checks whether the just-executed slice contained the secuentially last section, in which case data is pushed out from any variables marked as lastprivate to their shared counterparts outside the for construct. This concludes the 'infinite' loop.

After having exited from the 'infinite' loop, OdinMP/CCp restores the work-sharing construct information for the current thread (🅤), since it it now exiting from the current for construct. It then locks the team mutex again (🅥) and reduces its working copies of any reduction variables into the master copy, at 🅦. Likewise, it notes that it is exiting the current lap, and checks whether it might be the last thread in the team to do so (🅧); if it is, then this lap's information is no longer needed and can be deallocated. All team-critical work being done, the team's mutex is once more unlocked (🅨).

Finally, unless the 'nowait' clause is specified on the construct, OdinMP/CCp waits at 🅩 for all other threads in the team, by means of a barrier.

6.7 Single Construct

For the following code snippet

```
void foo ()
{
   int s;                    /* shared */
   int p;                    /* private */
   int fp = 0;               /* FirstPrivate */

   /* ... */
   #pragma omp parallel
   {
      #pragma omp single private(p) firstprivate(fp)
      {
         single_thread(&p, &fp);
      }
   }
   /* ... */
}
```

OdinMP/CCp generates the following code:

```
/*  # pragma omp  */  {
   /* private variables;
      if a variable is also firstprivate, then initialize it
      from the corresponding outer variable */
   int odinmp_flp_fp = (*odinmp_shared_data.par_0.fp);                 Ⓐ
   int p;

   /* odinmp variables */
   /* wsc info etc */                                                  Ⓑ
   odinmp_sing_slice_t odinmp_slice;
   odinmp_wsc_info_t odinmp_wsc_info;

   /* record info about this work sharing construct */                Ⓒ
   odinmp_wsc_info.next = ((odinmp_thread_info_t *) (odinmp_td))->wsc_info;
   ((odinmp_thread_info_t *) (odinmp_td))->wsc_info = &odinmp_wsc_info;
   odinmp_wsc_info.type = odinmp_wsc_type_sing;

   /* fetch a slice */
   {                                                                   Ⓓ
      odinmp_team_t *team = ((odinmp_thread_info_t *) (odinmp_td))->team;
      int lap;

      pthread_mutex_lock (&team->lock);                                Ⓔ
      lap = team->sing_laps[0];                                        Ⓕ
      if (!lap)
        lap = odinmp_td->sing_0.odinmp_lap;

      odinmp_slice.do_it = odinmp_td->sing_0.odinmp_lap >= lap;        Ⓖ
      team->sing_laps[0] = ++odinmp_td->sing_0.odinmp_lap;
      pthread_mutex_unlock (&team->lock);
   }                       /* fetch a slice */                         Ⓗ

   if (odinmp_slice.do_it) {                                           Ⓘ
      /* original code of single region comes here */
      {
        single_thread (&p, &odinmp_flp_fp);                            Ⓙ
      }
   }
   /* restore work sharing construct info */
   ((odinmp_thread_info_t *) (odinmp_td))->wsc_info = odinmp_wsc_info.next;  Ⓚ
   /* barrier at the end of single
      remove this section if nowait is specified */                    Ⓛ
   odinmp_barrier (((odinmp_thread_info_t *) (odinmp_td))->team,
         &((odinmp_thread_info_t *) (odinmp_td))->team->sing_barriers[0]);
}
```

This code is principally similar to that generated for for and sections constructs, but much simpler. At Ⓐ, a private copy of each private variable is allocated. Those that are also marked as firstprivate are initialized from the corresponding shared variable. At Ⓑ, OdinMP/CCp declares a variable to hold the information for the current slice to be run and information about this work-sharing construct. The work-sharing construct information is then initialized (Ⓒ).

Now, OdinMP/CCp prepares to fetch the slice (Ⓓ) for the current thread. To this purpose, it locks the team's mutex (Ⓔ) – to ensure that only one thread in this team eecutes the following code at a time. It then checks which lap in order is next supposed to have a thread execute the single construct (Ⓕ). OdinMP/CCp notes that permission to execute the code, is the same as asking whether *that* lap is the same as *this* lap (🅖). Finally, the team’s mutex is unlocked to let another team in the thread get at the lap descriptor (🅗).

At 🅘, the current thread checks its assigned slice to see if it should do any work, and if so, it executes the code (🅙). OdinMP/CCp restores the work-sharing construct information for the current thread (🅚), since it it now exiting from the constructf and finally, unless the ‘nowait’ clause is specified on the construct, waits at 🅛 for all other threads in the team, by means of a barrier.

6.8 *`Master`* *Construct*

A master construct such as

```
void foo ()
{
   /* ... */
   #pragma omp parallel
   {
      #pragma omp master
      {
         printf("only the master thread runs this");
      }
   }
   /* ... */
}
```

will be converted into

```
if (odinmp_is_master()) {                                                      🅐
   printf ("only the master thread runs this");                                🅑
}
```

This checks (by means of the `odinmp_is_master()` function, at 🅐) whether the calling thread is the master thread of its current team, and executes the code in the master region (🅑) if this is the case.

6.9 *`Critical`* *Construct*

A critical construct like

```
void foo ()
{
   /* ... */
   #pragma omp parallel
   {
      #pragma omp critical (bar)
      {
         printf("in critical region");
      }
   }
   /* ... */
}
```

will be converted into

```
pthread_mutex_lock (&odinmp_crit_bar);                                         🅐
{                                                                              🅑
   printf ("in critical region");
}                                                                                   Ⓒ
pthread_mutex_unlock (&odinmp_crit_bar);
```

Generally, if a name ‘bar’ is specified on the critical construct, then a mutex ‘odinmp_crit_bar’ is declared; otherwise the shared mutex ‘odinmp_crit_odinmp_default’ will be used. Either way, the mutex in question is locked (Ⓐ), then the critical code is executed (Ⓑ), at which point (Ⓒ) the mutex gets unlocked.

6.10 Barrier Directive

A barrier directive like

```
void foo ()
{
   /* ... */
   #pragma omp parallel
   {
      /* ... */
      #pragma omp barrier
      /* ... */
   }
   /* ... */
}
```

will be converted into

```
{
   odinmp_thread_data_t *odinmp_td = odinmp_self_data ();                          Ⓐ

   if (((odinmp_thread_info_t *) (odinmp_td))->team != ((void *) 0)) {             Ⓑ
      odinmp_barrier (((odinmp_thread_info_t *) (odinmp_td))->team,                Ⓒ
&((odinmp_thread_info_t *) (odinmp_td))->team->barriers[0]);
   }
}
```

First, the thread information for the current thread is fetched (Ⓐ). If the team information for the current thread is set, then we are currently executing in parallel (Ⓑ); and so, we call odinmp_barrier() (Ⓒ) for the appropriate barrier variable (one is declared for each barrier directive in the program) on the current team.

6.11 Atomic Construct

An atomic construct like

```
void foo ()
{
   int a;
   /* ... */
   #pragma omp parallel
   {
      /* ... */
      #pragma omp atomic
      a += 3;
      /* ... */
   }
   /* ... */
}
```

becomes

```
pthread_mutex_lock (&odinmp_atomic_lock);                                🅐
(*odinmp_shared_data.par_0.a) += 3;                                      🅑
pthread_mutex_unlock (&odinmp_atomic_lock);                              🅒
```

This simply brackets the statement at 🅑 in a critical region defined by locking (🅐) and unlocking (🅒) the global mutex `odinmp_atomic_lock`.

6.12 `Flush` Directive

The Flush directive, used as below:

```
#pragma omp flush
```

is transformed into the following code:

```
odinmp_sync_memory();
```

The implementation of the `odinmp_sync_memory()` function is platform-dependant, since it must ask the underlying platform to synchronize the current thread's view of memory with the global view.

`odinmp_sync_memory()` is also called implicitly from , among other places, within `odinmp_barrier()`, to implement the implicit flush specified in the OpenMP specification.

6.13 `Ordered` Construct

An ordered construct like the following

```
#pragma omp ordered
{
   printf ("ordered execution within parallel for loop");
}
```

becomes

```
{
   odinmp_thread_data_t *odinmp_td = odinmp_self_data ();                  🅐

   if (((odinmp_thread_info_t *) (odinmp_td))->wsc_info != ((void *) 0) &&
       ((odinmp_thread_info_t *) (odinmp_td))->wsc_info->type ==            🅑
odinmp_wsc_type_for) {

      odinmp_wsc_info_t *wsc_info = (((odinmp_thread_info_t *) (odinmp_td))-  🅒
>wsc_info);
      odinmp_for_lap_list_t *lap_list = wsc_info->u.wsc_for.lap_list;       🅓

      pthread_mutex_lock (&lap_list->for_data.order_lock);                  🅔
      while (lap_list->for_data.in_order != wsc_info->u.wsc_for.slice-      🅕
>in_order) {
        pthread_cond_wait (&lap_list->for_data.new_order, &lap_list-        🅖
>for_data.order_lock);
      }
      pthread_mutex_unlock (&lap_list->for_data.order_lock);                🅗
   }
}

/* original code here */
{
  printf ("ordered execution within parallel for loop");                           🅘
}
```

Since an ordered construct must only be called from dynamically within a for construct, OdinMP/CCp first fetches the current thread's information (🅐) and checks that this is the case (🅑). If it is, then it fetches the work-sharing construct information for closer inspection (🅒), as well as the current lap descriptor (🅓). Then the current thread locks the ordering mutex (🅔). While the ordering information indicates that it is not the current thread's turn yet (🅕), it waits for its turn to arrive (🅖), after which it unlocks the ordering mutex (🅗). Finally, at 🅘, the code is executed.

7 Building the OdinMP/CCp Compiler

The OdinMP/CCp compiler accepts full ANSI C with OpenMP extensions as input, and generates ANSI C with calls to the POSIX threads library, a.k.a. pthreads. In this chapter, I will offer a cursory overview over the construction of the OdinMP/CCp compiler. I will not go into any great detail, however.

7.1 Tools used to build the Automated Parallelizer

7.1.1 JavaCC: The Java Compiler Compiler

When searching for a foundation upon which to build OdinMP/CCp, I needed a good 'compiler compiler' or similar. Having had prior experience with yacc and lex, I searched for something different, and found JavaCC [7], The Java Compiler Compiler, from SunTest (distributed today by Metamata, Inc.). This is a simple yet powerful LL(1) parser generator, which also happened to include a complete grammar for ANSI C. I decided to use this as a starting point.

7.1.2 JTB: Java Tree Builder

Java Tree Builder (JTB) [8], 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. Each node class implements access methods for visitors. A skeleton depth-first visitor class, which walks through the whole node tree, is also generated; this can be used as a starting point for writing other visitors.

The approach supported by JTB suited me very well, since it meant that the changes to the grammar could be separated from the code which actually worked with the parsed program – I could place this in a visitor specific to the task.

7.2 The C Grammar

The C grammar I use is basically the sample C grammar distributed with JavaCC, but adapted to support OpenMP directives and constructs.

7.3 Supporting Classes

Besides writing a suitable visitor class to operate on the parsed OpenMP-annotated ANSI C code, I had to write a small framework of supporting classes which help me build and maintain the data structures I need to generate the code. I will present a brief overview of these classes below.

7.3.1 Decl

A `Decl` handles collecting type and modifier information in a C declaration, and is used to construct the `Symbol`s declared.

7.3.2 Symbol

A `Symbol` holds a declared symbol's name and type information, as well as convenience methods for acessing and changing them.

7.3.3 SymbolTable

Not surprisingly, a `SymbolTable` holds associations between names and their respective symbols.

7.3.4 OmpSymbolAttributes

Each symbol may be 'marked' with a number of attributes by OpenMP – shared or private, for instance – and these attributes are stored in instances of the `OmpSymbolAttributes` class, one for each affected symbol on the construct that so marks a symbol.

7.3.5 ConstructDescriptor and its Subclasses

For each OpenMP construct and directive, I have written a corresponding `ConstructDescriptor` subclass, which holds all the pertinent information about the construct, and can generate the appropriate code. There is a `ParallelConstructDescriptor` to handle a parallel construct, a `ForConstructDescriptor` that handles a for construct, and so on. Each `ConstructDescriptor` also keeps track of which symbols it marks with any particular `OmpSymbolAttributes`.

7.3.6 ClauseHandler and its Subclasses

I found it useful to factor out the handling of clauses on OpenMP constructs, from the handling of the constructs themselves. So, for each construct than handles clauses, there exists a *`Construct`*`ClauseHandler` class which sets up the *`Construct`*`ConstructDescriptor` according to the clauses specified. The usefulness of this is especially apparent when considering a combined parallel work-sharing construct such as the parallel for construct, such as a parallel for construct. In this case, OdinMP/CCp generates both a `ForConstructDescriptor` and a `ParallelConstructDescriptor`, and uses a `ParallelForClauseHandler` to distribute the clauses between the two as appropriate.

7.4 The DataBuildingVisitor

The `DataBuildingVisitor` class is slightly misnamed; I initially thought I would use two different visitors, one to build the data necessary to generate the code and another one to actually generate the code, but as it turned out I only needed one. Since I am reluctant to introduce new bugs into my code, I have allowed it to keep its name, slightly misrepresenting as it is.

Basically, the `DataBuildingVisitor` walks through the node tree, parsing the C code, building type and symbol tables, and at the same time generating equivalent code as its result. When it encounters an OpenMP directive or construct, a suitable `ConstructDescriptor` (or, in the case of combined parallel work-sharing constructs, two `ConstructDescriptor`s) is created, as well as any necessary `ClauseHandler`. Then that `ConstructDescriptor` is asked to supply the code for itself.

7.5 The OdinMP class

To pull this all together, the `OdinMP` class contains the main method to run the OdinMP/CCp compiler, fetching arguments from the command line or passed as Java properties. It then parses each source file specified, and lets a `DataBuildingVisitor` walk through the parse tree. Then, for each input source file a corresponding output file is generated, and finally the shared `odinmp.c` file is generated as well.

8 Measurements, Results and Other Lessons

Code equivalent to that generated by OdinMP/CCp, but written by hand prior to the development of the OdinMP/CCp compiler, has been evaluated regarding performance on a small number of different platforms; these results are, however, too informal to be presented here. Generally, OdinMP/CCp-compiled code *will* make use of the multiprocessing capabilities of the underlying platform to a useful extent. However, as can be seen in section 6, OdinMP/CCp generates quite a bit of code to handle OpenMP constructs. So, in order for OdinMP/CCp to give measurable improvements, the execution time of the code inside the constructs should be larger than that used by the OdinMP/CCp added code. Basically, for small problems, I expected OdinMP/CCp to offer little, if any, speedup, whereas as the problem domain increased, OdinMP/CCP's overhead would shrink in comparison to what was gained.

After I finished development of the OdinMP/CCp compiler, I have been able to perform measurements on some of the actual code generated by OdinMP/CCp.

8.1 Speedup

Overall speedup is, of course, one of the most interesting and important figures. To measure this, I took the Molecular Dynamics sample program [9] from the OpenMP ARB web site and translated it from FORTRAN to C. The resulting program is included in Appendix B. It was run for a dataset of 2048 particles, both in simple sequential execution – without any involvement from OdinMP/CCp, i.e. with no parallelism at all – as well as parallelized by OdinMP/CCp, for a variety of numbers of processors, on a SUN Enterprise 10000 (using 250MHz SPARC processors). Much to my pleasure, the OdinMP/CCP-compiled version exhibits near-linear speedup, achieving a speedup of 7.2 on 8 processors. The full results of the run are tabulated in Table 1, and plotted in Figure 10.

ideal	OdinMP/CCp
1	1.00
2	1.96
3	2.90
4	3.86
5	4.79
6	5.39
7	6.67
8	7.28

Table 1, Speedup, md, Sun Enterprise 10000

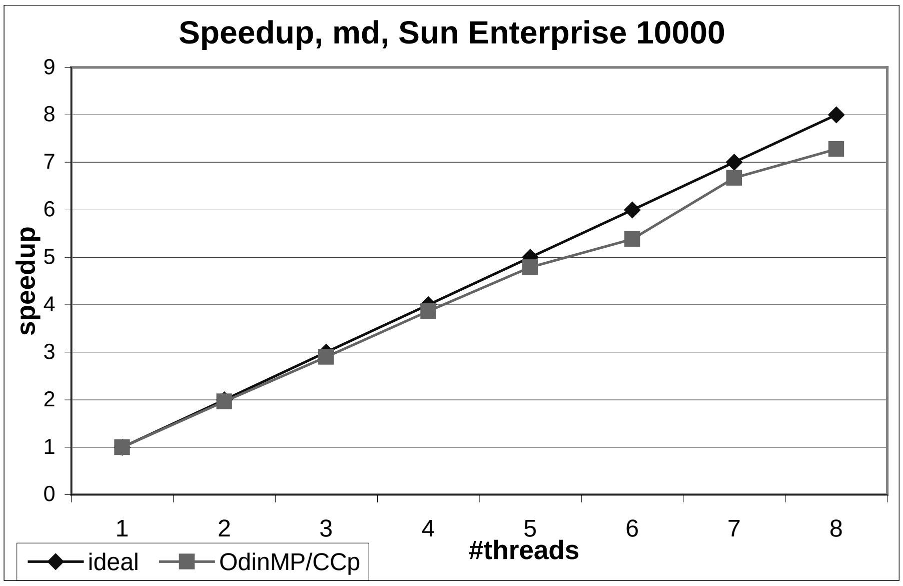


Figure 10, Speedup, md, Sun Enterprise 10000

I also ran the same program on a SGI Origin 2000 with MIPS R12000 processors at 300 MHz, comparing the resulting execution times and speedups that could be achieved with different combinations of C compiler and OpenMP implementation. On the Origin I had access to SGI's 'cc' compiler, which implements OpenMP itself, as well as 'gcc', the GNU C compiler. So, I ran the same problem (md, with 2048 particles) for the combinations:

1. SGI cc, no parallelism
2. GNU gcc, no parallelism
3. SGI cc, SGI OpenMP
4. SGI cc, OdinMP
5. GNU gcc, OdinMP

All compilations were done with –O3 optimization. The resulting execution times are Tabulated in Table 2, and plotted in Figure 11.

#threads	cc	gcc	cc-omp	cc-odinmp	gcc-odinmp
1	32.80	28.67	33.36	42.17	33.36
2	32.80	28.67	16.81	21.09	16.71
3	32.80	28.67	11.24	14.17	11.40
4	32.80	28.67	8.44	10.59	8.39
5	32.80	28.67	6.80	8.53	6.74
6	32.80	28.67	5.97	7.13	5.81
7	32.80	28.67	4.97	6.24	5.15
8	32.80	28.67	4.39	5.36	4.28

Table 2, Execution Times (s), md, SGI Origin 2000

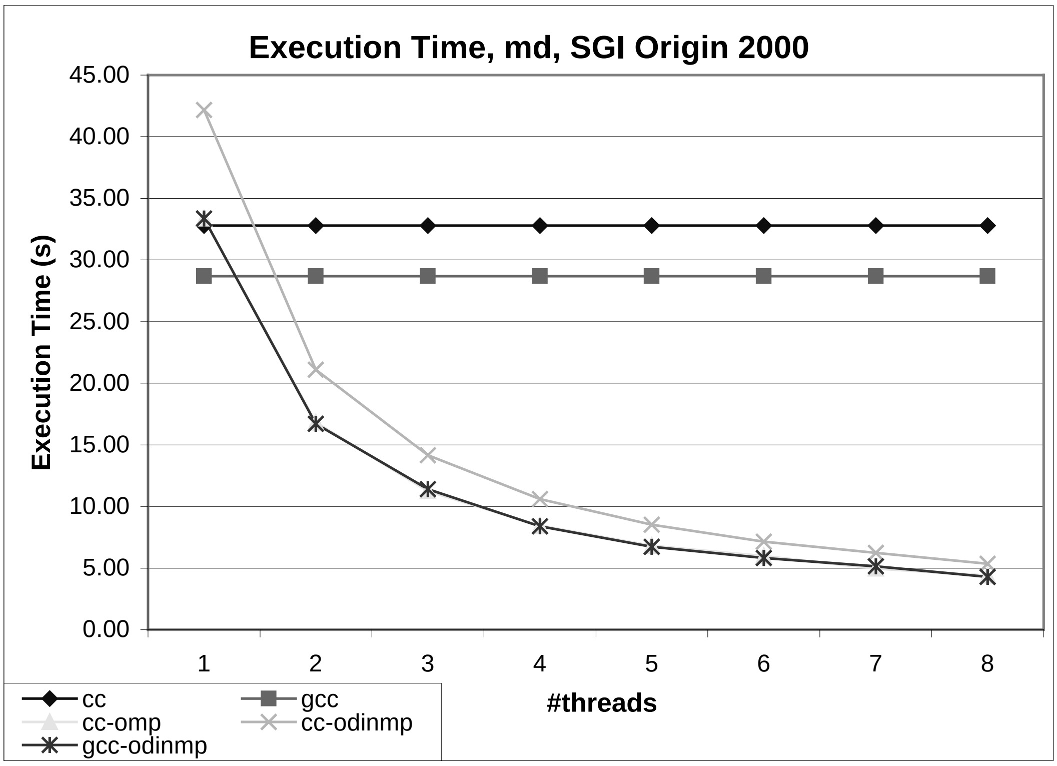


Figure 11, Execution Times, md, SGI Origin 2000

One of the things we can see from these figures is that the combination of OdinMP and gcc results in almost exactly the same execution times and speedup as are offered by SGI's own OpenMP compiler. In fact, if we plot the speedups (in this case, the execution time with one process / the execution time with n processes, for the same compiler combination) for the three OpenMP variations above, we get the figures in Table 3 and the graph in Figure 12.

ideal	cc-omp	cc-odinmp	gcc-odinmp
1	1.00	1.00	1.00
2	1.98	2.00	2.00
3	2.97	2.98	2.93
4	3.95	3.98	3.97
5	4.90	4.94	4.95
6	5.59	5.91	5.74
7	6.71	6.76	6.47
8	7.60	7.87	7.80

Table 3, Speedups, md, SGI Origin 2000

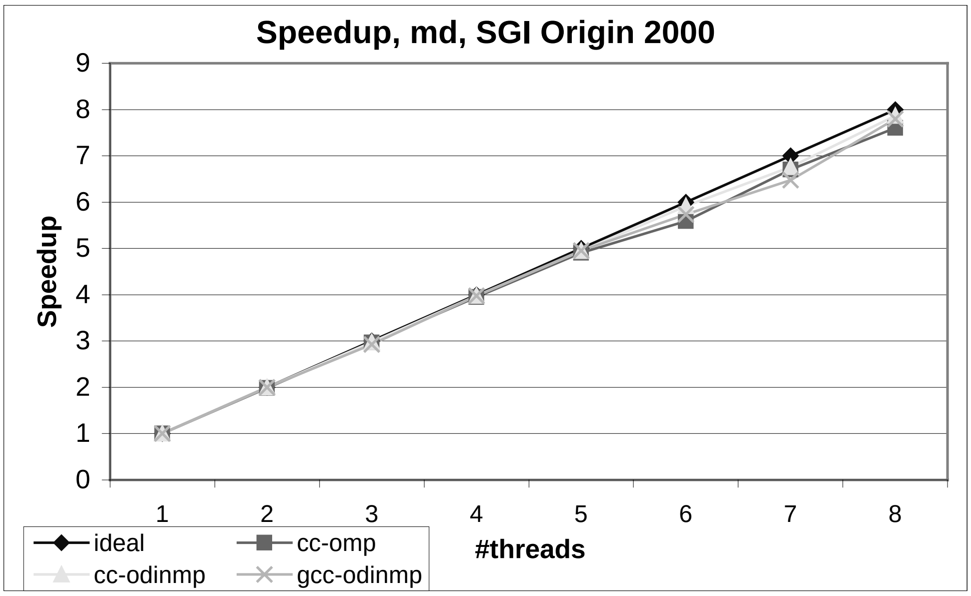


Figure 12, Speedups, md, SGI origin 2000

The offered data suggests that OdinMP indeed makes good use of the multiprocessing capabilities offered by the underlying platform, on two such different systems as a bus-based shared-memory machine (the SUN Enterprise 10000) and a CC-NUMA machine (the SGI Origin 2000).

8.2 Overhead

For completeness, I attempted to measure the overhead injected by OdinMP at the beginning and end of a parallel region, and compare this to similar overhead injected by SGI's OpenMP compiler. For this purpose, I wrote a small program which measures the time taken to execute a piece of code both for the case where the code in question is inside a parallel region, and the case where it is in a sequential region. Code for this program is included in Appendix C.

I ran a small test suite, measuring the overhead as reported by my program as compiled with SGI's cc using its OpenMP implementation, and GNU gcc with OdinMP, respectively, for different numbers of threads. Measurements were repeated several times, and the lowers overhead figures selected, to give the results tabulated in Table 4 and plotted in Figure 13.

Overhead (μs)		
#threads	cc-omp	gcc-odinmp
1	54.55	1.81
2	79.94	102.26
3	102.77	135.41
4	122.22	191.06
5	145.62	235.00
6	142.12	294.72
7	135.22	334.23
8	211.52	376.64

Table 4, Parallel Region Overhead

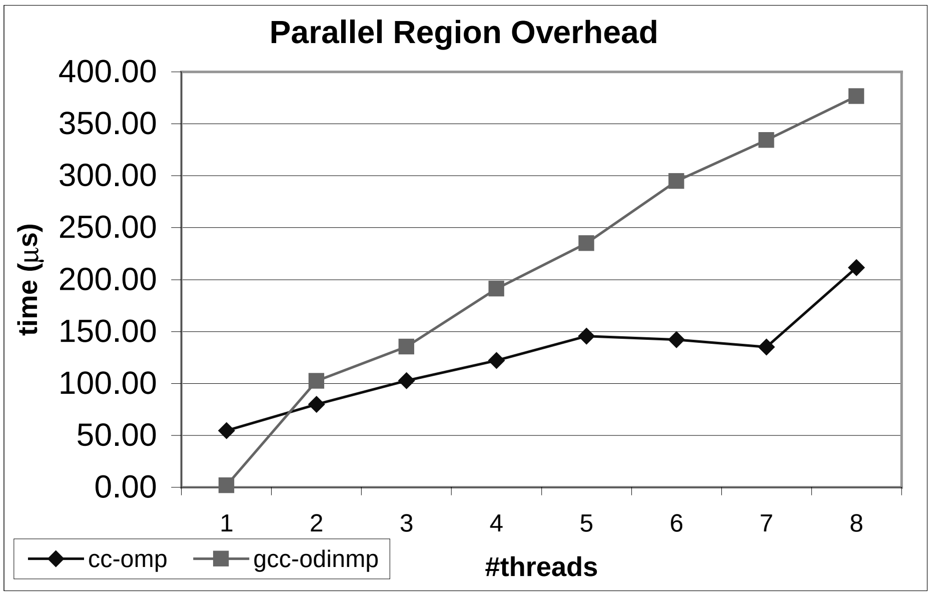


Figure 13, Parallel Region Overhead

One can note that for the case of only 1 thread, OdinMP's overhead is less than that generated by SGI's OpenMP implementation – alas, this is of course largely irrelevant when the goal is to make use of multiple processors. However, even for greater numbers of threads, OdinMP's overhead is within approximately a factor of 2, compared to SGI's cc. Considering that OdinMP does not have the luxury of making any assumptions whatsoever about the underlying platform, I find these figures to be quite encouraging.

9 Conclusion

The purpose of this project was to investigate the viability, and if possible produce an implementation, of a C-to-C compiler for OpenMP. It would take OpenMP-annotated C code as input and generate C code with calls to the POSIX threads library 'pthreads' as output, and it would be portable and usable on different platforms, while giving reasonable performance. In the process I had to find a suitable set of tools with which to write the compiler.

The result that I lay before you is the OdinMP/CCp compiler, which achieves all of the above. It offers a conformant implementation of the OpenMP standard for the C programming language. It will effortlessly port to most modern Unix systems, and can be ported to any other platform rather painlessly, by virtue of being written in platform-independent Java. The code generated by OdinMP/CCp performs reasonably similarly to code generated by a commercial OpenMP implementation.

Appendix A Generated Code

A.1 Common Types

The following types are made available to the generated code:

```
/* Common stuff for all parsed files follows! */
/* Generated private & shared memory data types: */
#define ODINMP_NUM_THREADS 4
#define ODINMP_NUM_PROCESSORS 4
```

These define the 'number of processors' which OdinMP/CCp will report, as well as the default number of threads to be used unless overridden by the OMP_NUM_THREADS environment variable.

```
typedef struct _odinmp_team_t odinmp_team_t;
typedef struct _odinmp_thread_data_t odinmp_thread_data_t;
typedef struct _odinmp_shared_data_t odinmp_shared_data_t;
```

These will be used later.

```
typedef struct {
  long long int lb;
  long long int incr;
  long long int b;
  int last, done;
  int i;
  int in_order;
} odinmp_for_slice_t;
```

This will hold information about which slice of a for loop each thread is assigned during parallel execution of a for loop.

```
typedef enum {
  odinmp_scheduler_runtime = 0,
  odinmp_scheduler_static = 1,
  odinmp_scheduler_dynamic = 2,
  odinmp_scheduler_guided = 3
} odinmp_scheduler_type_t;
```

This identifies the runtime scheduler type.

```
typedef struct {
  odinmp_scheduler_type_t type;
  double chunk_size;
} odinmp_scheduler_t;
```

This holds both type and chunk size information about the runtime-selected scheduler.

```
typedef enum {
  odinmp_barrier_state_empty = 0,
  odinmp_barrier_state_waiting_for_join,
  odinmp_barrier_state_waiting_for_depart
} odinmp_barrier_state_t;
```

This holds information about the state of a barrier.

```
typedef struct {
  pthread_cond_t all_joined;
  pthread_cond_t all_departed;
  int n_left_to_join;
  int n_left_to_depart;
  odinmp_barrier_state_t state;
} odinmp_barrier_t;
```

This defines the barrier itself: condition variables for when all threads have joined, and when all threads have departed from the barrier, respectively, as well as counters that keep track of state transitions.

```
typedef struct {
  long long int lb;
  long long int incr;
  long long int b;
  double chunk_size;
  odinmp_scheduler_t scheduler;

  pthread_mutex_t order_lock;
  pthread_cond_t new_order;
  int in_order;
  int next_in_order;

} odinmp_for_data_t;
```

This holds all the pertinent information about a for construct: The initial value, the final value, and the increment; the chunk size, the scheduler to use; and variables used with the 'ordered' clause and 'ordered' construct.

```
typedef struct _odinmp_for_lap_list_t {
  odinmp_for_data_t for_data;
  int lap;
  int n_left_to_leave;

  struct _odinmp_for_lap_list_t *next;
} odinmp_for_lap_list_t;
```

This holds information about one lap of execution of a for construct.

```
typedef struct {
  int section;
  int last;
  int done;
} odinmp_sect_slice_t;
```

Slice information for a sections construct.

```
typedef struct {
  int n_left;
} odinmp_sect_data_t;
```

Information about one sections construct.

```
typedef struct _odinmp_sect_lap_list_t {
  odinmp_sect_data_t sect_data;
  int lap;
  int n_left_to_leave;

  struct _odinmp_sect_lap_list_t *next;
} odinmp_sect_lap_list_t;
```

This holds information about one lap of execution of a sections construct.

```
typedef struct {
  int do_it;
} odinmp_sing_slice_t;
```

The slice for a single construct

```
typedef struct _odinmp_team_list_t {
  odinmp_team_t *team;
  struct _odinmp_team_list_t *next;
} odinmp_team_list_t;
```

A list of teams.

```
enum {
  odinmp_wsc_type_none = 0,
  odinmp_wsc_type_for,
  odinmp_wsc_type_sect,
  odinmp_wsc_type_sing
} odinmp_wsc_type_t;
```

To keep track of what current work-sharing construct is being executed.

```
typedef struct _odinmp_wsc_info_t {
  odinmp_wsc_type_t type;
  struct _odinmp_wsc_info_t *next;

  union {
    struct {
      odinmp_for_lap_list_t *lap_list;
      odinmp_for_slice_t *slice;
    } wsc_for;

    struct {
      odinmp_sect_lap_list_t *lap_list;
    } wsc_sect;
  } u;
} odinmp_wsc_info_t;
```

Work sharing construct information: what type it is, and data for the different types of work sharing constructs.

```
typedef enum {
  odinmp_thread_state_waiting_for_work,
  odinmp_thread_state_has_work_to_do,
  odinmp_thread_state_working,
  odinmp_thread_state_finished_work,
  odinmp_thread_state_exiting
} odinmp_thread_state_t;
```

The different states that a thread can be in.

```
typedef struct {
  pthread_mutex_t lock;
  pthread_cond_t work_available;
  pthread_cond_t finished_work;
  pthread_cond_t stopped_spinning;
  pthread_t pthread_id;
  odinmp_thread_state_t state;

  int id;
  int run_region;
  odinmp_wsc_info_t *wsc_info;

  int tdepth;

  odinmp_team_t *team;
  odinmp_team_list_t *teams;
} odinmp_thread_info_t;
```

Thread information: a lock, condition variables to signal certain thread states, id, pthread id, information about what parallel region to run, a pointer to whatever work sharing construct it might be running, and pointers to the current topmost team, and a list of other teams the thread might be in.

A.2 Generated types

The following types are generated:

```
/* typedef */ struct _odinmp_thread_data_t {
   odinmp_thread_info_t thread_info;

   /* A */                                                              🅐

};
```

Each parallel region and work sharing construct will add whatever private data they need, at 🅐.

```
/* typedef */ struct _odinmp_shared_data_t {

   /* A */                                                              🅐

};
```

At 🅐, each parallel region will add a struct of pointers to such shared variables as are not globally visible within the file.

```
/* typedef */ struct _odinmp_team_t {
   pthread_mutex_t lock;

   int id;
   int *ids;
   int n;

    /* A */                                                             🅐

} /* odinmp_team_t */ ;
```

At 🅐, for each work sharing construct, we add a barrier and a lap descriptor list pointer.

A.3 Externally Declared Symbols

The following external symbols are available for use by OdinMP/CCp code:

```
/* External declarations for OdinMP/CCp: */
extern odinmp_team_t odinmp_main_team;
```

Team identifier for the main team.

```
extern odinmp_shared_data_t odinmp_shared_data;
```

Holds pointers to shared variables.

```
extern odinmp_thread_data_t **odinmp_thread_datas;
```
Holds an array of pointers to thread data, one for each thread.

```
extern int odinmp_num_threads;
```
How many threads are currently running.

```
extern int odinmp_num_threads_allocated;
```
How many threads are currently allocated.

```
extern int odinmp_debug;
```
Should we print debug information?

```
extern int odinmp_timing;
```
Should we print timing information?

```
extern pthread_key_t odinmp_pthread_key;
```
The pthread key with which we identify OdinMP/CCp's thread-specific data.

```
extern odinmp_scheduler_type_t odinmp_runtime_scheduler_type;
```
Identifies the runtime scheduler chosen by the user

```
extern void odinmp_printf(char *fmt, ...);
```
For printing debug messages.

```
extern void odinmp_thread(odinmp_thread_data_t *odinmp_td);
```
The global thread action function.

```
extern void odinmp_thread_init(odinmp_thread_data_t *td, int i);
```
Initializes a thread.

```
extern odinmp_thread_data_t *odinmp_self_data();
```
Returns the thread-specific data for this thread.

```
extern void odinmp_start_thread(odinmp_thread_data_t *td);
```
Asks OdinMP/CCp to start a new thread.

```
extern void odinmp_stop_thread(odinmp_thread_data_t *td);
```
Asks OdinMP/CCp to stop a thread.

```
extern int odinmp_get_num_threads(void);
```
Asks OdinMP/CCp how many threads are currently running.

```
extern void odinmp_set_num_threads(int num_threads);
```
Sets the number of threads for OdinMP/CCp to use.

```
extern odinmp_thread_data_t *odinmp_allocate_thread_data();
```
Allocates thread-specific data for one thread.

```
extern odinmp_team_t *odinmp_allocate_team(int max_num_threads);
```
Allocates a team of threads.

```
extern int odinmp_get_thread_num();
```
Returns the current thread's number.

```
extern int odinmp_is_master();
```
Returns wether the current thread is the master thread of its topmost team.

```
extern void odinmp_dispatch(odinmp_thread_data_t *td);
```
Tells OdinMP/CCp to tell a specific thread to run a specified parallel region.

```
extern void odinmp_free_team(odinmp_team_t *team);
```
Deallocates a team of threads.

```
extern void odinmp_wait_for_work(odinmp_thread_data_t *td);
```
A request for something to do from an idle thread.

```
extern void odinmp_finished_work(odinmp_thread_data_t *td);
```
Notifies OdinMP/CCp that the calling thread has finished executing its parallel region.

```
extern void odinmp_wait_finished(int id);
```
Waits until the thread specified by id has finished work.

```
extern void odinmp_thread_spin(odinmp_thread_data_t *td);
```
The global thread spinning function, which repeatedly waits for work, executes the specified parallel region, then tells the system that work has been finished.

```
extern void *odinmp_thread_pthread(void *data);
```
The function which is handed to pthread_create, and which contains the whole life cycle of a thread.

```
extern void odinmp_init ();
```
Initializes OdinMP/CCp.

```
extern void odinmp_atexit();
```
Cleans up after OdinMP/CCp.

```
extern void odinmp_complain();
```
Complains about erroneous conditions during program execution.

```
extern int odinmp_get_num_procs();
```
Returns the number of processors for which this program was compiled.

```
extern void odinmp_barrier(odinmp_team_t *team, odinmp_barrier_t *b);
```
Implements a barrier; waits until all threads in team have reached the barrier, and only then lets any of them proceed.

```
extern void odinmp_sync_memory();
```
Synchronizes the calling thread's view of memory with the global one. Used for flush.

```
extern void odinmp_parse_env ();
```
Parses the OpenMP environment variables (OMP_NUM_THREADS, OMP_SCHEDULE) and tells OdinMP/CCp about their values.

```
extern void odinmp_parse_args (int *argc, char **argv);
```
Parses command-line arguments for OdinMP/CCp use.

Appendix B   Molecular Dynamic Program

B.1   md.c

```
/***********************************************************************
 * This program implements a simple molecular dynamics simulation,
 *   using the velocity Verlet time integration scheme. The particles
 *   interact with a central pair potential.
 *
 * Author:   Bill Magro, Kuck and Associates, Inc. (KAI), 1998
 *
 * Parallelism is implemented via OpenMP directives.
***********************************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include "omp.h"
#ifndef RAND_MAX
#define RAND_MAX 0x7fff
#endif

#define ndim 3
#define nparts 2048
#define NSTEPS 20

#undef LONGDOUBLE
#ifdef LONGDOUBLE
typedef long double real8;
#define R8F "% 15.7Le"
#else
typedef double real8;
#define R8F "% 15.7e"
#endif /* LONGDOUBLE */

double t() {
  struct timeval tv;
  gettimeofday(&tv, ((void *)0));
  return (double)tv.tv_sec + (double)tv.tv_usec/1000000.0;
}

typedef real8 vnd_t[ndim] ;

/* statement function for the pair potential and its derivative
   This potential is a harmonic well which smoothly saturates to a
   maximum value at PI/2.  */

real8 v(real8 x) {
  if (x < M_PI_2)
    return pow(sin(x), 2.0);
  else
    return 1.0;
```

```
}

real8 dv(real8 x) {
  if (x < M_PI_2)
    return 2.0 * sin(x) * cos(x);
  else
    return 0.0;
}


/**********************************************************************
*
 * Initialize the positions, velocities, and accelerations.

**********************************************************************
/
void initialize(int np, int nd,
            vnd_t box, vnd_t *pos, vnd_t *vel, vnd_t *acc)
{
  int i, j;
  real8 x;

  srand(4711L);
  for (i = 0; i < np; i++) {
    for (j = 0; j < nd; j++) {
      x = rand() % 10000/(real8)10000.0;
      pos[i][j] = box[j]*x;
      vel[i][j] = 0.0;
      acc[i][j] = 0.0;
    }
  }
}

/* Compute the displacement vector (and its norm) between two
particles. */
real8 dist(int nd, vnd_t box, vnd_t r1, vnd_t r2, vnd_t dr)
{
  int i;
  real8 d;

  d = 0.0;
  for (i = 0; i < nd; i++) {
    dr[i] = r1[i] - r2[i];
    d += dr[i] * dr[i];
  }
  return sqrt(d);
}

/* Return the dot product between two vectors of type real*8 and length
n */
real8 dotr8(int n, vnd_t x,vnd_t y)
{
  int i;
  real8 t = 0.0;

  for (i = 0; i < n; i++) {
    t += x[i]*y[i];
  }

  return t;
}

/*********************************************************************
*
 * Compute the forces and energies, given positions, masses,
 * and velocities

*********************************************************************
/
void compute(int np, int nd,
           real8 *box,
           vnd_t *pos, vnd_t *vel,
           real8 mass, vnd_t *f,
           real8 *pot_p, real8 *kin_p)
{
  real8 x;
  int i, j, k;
  vnd_t rij;
  real8  d;
  real8 pot, kin;

  pot = 0.0;
  kin = 0.0;

  /* The computation of forces and energies is fully parallel. */
#pragma omp parallel for default(shared) private(i,j,k,rij,d)
reduction(+ : pot, kin)
  for (i = 0; i < np; i++) {
    /* compute potential energy and forces */
    for (j = 0; j < nd; j++)
      f[i][j] = 0.0;

    for (j = 0; j < np; j++) {
      if (i != j) {
      d = dist(nd,box,pos[i],pos[j],rij);
      /* attribute half of the potential energy to particle ’j’ */
      pot = pot + 0.5 * v(d);
      for (k = 0; k < nd; k++) {
        f[i][k] = f[i][k] - rij[k]* dv(d) /d;
      }
      }
    }
    /* compute kinetic energy */
    kin = kin + dotr8(nd,vel[i],vel[j]);
  }

  kin = kin*0.5*mass;

  *pot_p = pot;
  *kin_p = kin;
}

/*********************************************************************
*
```

```
 * Perform the time integration, using a velocity Verlet algorithm

************************************************************************
/
void update(int np, int nd, vnd_t *pos, vnd_t *vel, vnd_t *f, vnd_t *a,
          real8 mass, real8 dt)
{
  int i, j;
  real8 rmass;

  rmass = 1.0/mass;

  /* The time integration is fully parallel */
#pragma omp parallel for default(shared) private(i,j)
firstprivate(rmass, dt)
  for (i = 0; i < np; i++) {
    for (j = 0; j < nd; j++) {
      pos[i][j] = pos[i][j] + vel[i][j]*dt + 0.5*dt*dt*a[i][j];
      vel[i][j] = vel[i][j] + 0.5*dt*(f[i][j]*rmass + a[i][j]);
      a[i][j] = f[i][j]*rmass;
    }
  }
}

/******************
 * main program
 *****************/



int main (int argc, char **argv) {

  /* simulation parameters */

  real8 mass = 1.0;
  real8 dt = 1.0e-4;
  vnd_t box;
  vnd_t position[nparts];
  vnd_t velocity[nparts];
  vnd_t force[nparts];
  vnd_t accel[nparts];
  real8 potential, kinetic, E0;
  int i;
  int nsteps = NSTEPS;
  double t0, t1;

  if (argc == 2)
    nsteps = atoi(argv[1]);

  for (i = 0; i < ndim; i++)
    box[i] = 10.0;

  t0 = t();

  /* set initial positions, velocities, and accelerations */
  initialize(nparts,ndim,box,position,velocity,accel);
  /* compute the forces and energies */
  compute(nparts,ndim,box,position,velocity,mass,
        force,&potential,&kinetic);
  E0 = potential + kinetic;

  /* This is the main time stepping loop */
  for (i = 0; i < nsteps; i++) {
    compute(nparts,ndim,box,position,velocity,mass,
          force,&potential,&kinetic);
    printf(R8F " " R8F " " R8F "\n",
         potential, kinetic, (potential + kinetic - E0)/E0);
    update(nparts,ndim,position,velocity,force,accel,mass,dt);
  }

  t1 = t();
  printf("execution time with %d threads: %lf s\n",
#ifdef _OPENMP
  omp_get_max_threads(),
#else
  1,
#endif
  t1 - t0);

  exit (0);
}
```

Appendix C   Overhead Measurement Program

C.1   *nop.c*

```
void *nop(void *p) {
  return p;
}
```

C.2   *spin.c*

```
extern void *nop (void *);

extern int spin_factor;

void spin(double jLimit) {
  int i;
  double j;
  double d;
  for (i = 0; i < spin_factor; i++)
    for (j = 0.0; j < jLimit; j += 1.0)
      nop(&d);
}
```

C.3   *overhead.c*

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

int spin_factor = 1000;

extern void spin (double);

double t() {
    struct timeval tv;
    gettimeofday(&tv, ((void *)0));
    return (double)tv.tv_sec + (double)tv.tv_usec/1000000.0;
}

int main (int argc, char **argv) {
  int i;
  double t0, t1, dt1t0, t2, t3, dt3t2;
  int num = 100;
  double dt1t0_iter, dt3t2_iter;
  double diff, diff_iter;

  for (i = 1; i < argc; i++) {
    if (!strcmp(argv[i], "-n")) {
      if (argc <= (i+1)) {
      fprintf(stderr,"need argument to ’-n’\n");
      exit(1);
      }
      num = atoi(argv[++i]);
    } else if (!strcmp(argv[i], "-f")) {
      if (argc <= (i+1)) {
      fprintf(stderr,"need argument to ’-f’\n");
      exit(1);
      }
      spin_factor = atoi(argv[++i]);
    }

  }

  printf("Parallel:\n");

  t0 = t();
  for (i = 0; i < num; i++) {
#pragma omp parallel
      spin(1000.0);
  }
  t1 = t();
  dt1t0 = t1 - t0;

  printf ("time             : %15.9g s = %15.9g ms = %15.9g us\n",
        dt1t0, dt1t0 * 1000.0, dt1t0 * 1000000.0);

  dt1t0_iter = dt1t0 / num;

  printf ("time / iteration : %15.9g s = %15.9g ms = %15.9g us\n",
        dt1t0_iter, dt1t0_iter * 1000.0, dt1t0_iter * 1000000.0);


  printf ("\nSequential:\n");
  t2 = t();
  for (i = 0; i < num; i++) {
      spin(1000.0);
  }
  t3 = t();
  dt3t2 = t3 - t2;

  printf ("time             : %15.9g s = %15.9g ms = %15.9g us\n",
        dt3t2, dt3t2 * 1000.0, dt3t2 * 1000000.0);

  dt3t2_iter = dt3t2 / num;

  printf ("time / iteration : %15.9g s = %15.9g ms = %15.9g us\n",
        dt3t2_iter, dt3t2_iter * 1000.0, dt3t2_iter * 1000000.0);

  printf("\n");
  diff = dt1t0 - dt3t2;
  diff_iter = dt1t0_iter - dt3t2_iter;

  printf ("overhead         : %15.9g s = %15.9g ms = %15.9g us\n",
        diff, diff * 1000.0, diff * 1000000.0);

  printf ("overhead / iter. : %15.9g s = %15.9g ms = %15.9g us\n",
        diff_iter, diff_iter * 1000.0, diff_iter * 1000000.0);
}
```

Appendix D Installing and Using OdinMP/CCp

This Appendix reproduces the HTML document which describes how to install and use OdinMP/CCp.

Installing and Using OdinMP/CCp

Prerequisites

These installation instructions assume that you have fetched the OdinMP/CCp archive, `odinmp-1.0.tar.gz`.

Installing

1. Fetch the OdinMP/CCp archive and place it in a temporary location, such as ***tempdir*/`odinmp-1.0.tar.gz`** .
2. To install OdinMP/CCp, choose where you wish to install OdinMP/CCp - call this ***install-root*** . A good choice for install-root would be, for instance, `/usr/local` , `/opt` , or perhaps your home directory.
3. **`cd`** to ***install-root***
4. Unpack the OdinMP/CCp archive:
   `gzcat tempdir/odinmp-1.0.tar.gz | tar -xvf -`
   his will create a subdirectory `odinmp` beneath the current directory, which contains all the necessary OdinMP files.

You have now unpacked the OdinMP/CCp files.

Mandatory Configuration

Before you can begin using OdinMP/CCp, you must inform the OdinMP/CCp system where it is installed:

1. Open the file ***install-root*/`odinmp/bin/odinmp_prep`** in a text editor. Near the top, there is an area which reads something like
   ```
   #################################################################
   # Edit this to point to the directory where you installed
   # OdinMP/CCp
   #
   $odinmp_home = "INSTALLDIR/odinmp";
   #
   #################################################################
   ```
   Replace 'INSTALLDIR' (or whatever is in its place) with '***install-root***' . If your *install-root* is **`/usr/local`**, then the line should read exactly
   `$odinmp_home = "/usr/local/odinmp";`
2. Similarly, open the file ***install-root*/`odinmp/bin/OdinMP`**, and perform the *same* editing actions on that as you did with **`odinmp_prep`** above.

Requirements and Optional Configuration

To run, OdinMP/CCp requires two things.

Perl version 5

The programs in ***install-root*/`odinmp/bin/`** are Perl version 5 programs. They expect Perl version 5 to be installed somewhere in your search path, so that it can be found by /usr/bin/env .

If the programs won't immediately run on your system, modify the top line of each program to point to the correct location for Perl version 5.

Java 1.1 or newer

A Java runtime, version 1.1 or newer, which must be in the program search path, such as identifed by the PATH environment variable. This is used by the ***install-root/*****`odinmp/bin/OdinMP`** program. If necessary, you can edit ***install-root/*****`odinmp/bin/OdinMP`** to explicitly point out the java interpreter you require.

Running OdinMP/CCp

Consider that you have a program `foobar`, compiled from files `foo.c` and `bar.c`, which you wish to parallelize through OdinMP/CCp. Here is a description, step by step, of what you need to do.

`cd` to the directory containing the original source files

This places you in the directory which contains the original source files for your program foo.c and bar.c .

Prepare for OdinMP/CCp parallelization

Run the command
`odinmp_prep --output=`*`foobar`*` --targetDirectory=`*`foobar_parallel foo.c bar.c`*
This will create a directory called **`foobar_parallel`**, and in that directory a `makefile` and symbolic links to the files '**`omp.c`**' and '**`omp.h`**' as supplied with OdinMP/CCp.

`cd` to foopar_parallel

This places you in the same directory as the generated `makefile`.

`make`

This will call upon OdinMP/CCp to walk through the files specified on the **odinmp_prep** command line, parallelizing each one, and generating a corresponding output file in the current directory. Each generated C source file will be compiled, and finally all object files get linked to form the resulting program.

A Simple Example

The NAS benchmark suite for OpenMP includes a program called 'laplace'. To build this program, a Makefile is provided:

```
#
# Makefile for Laplace
#

# OPT = -g
# OPT = -xO4

OBJ = lap.o
INCL=lap.h

# CC=gcc
CFLAGS  = $(OPT)
```

```
LDLIBS = -lm
OMPCC = ompcc

PROGRAM = lap-omp lap-seq

all: $(PROGRAM)

lap-omp: lap.c second.o
        $(OMPCC) $(CFLAGS) -o lap-omp lap.c second.o $(LDLIBS)

lap-seq: lap.c second.o
        $(CC) $(CFLAGS) -o lap-seq lap.c second.o $(LDLIBS)

clean:
        rm -f $(PROGRAM)
```

Since OdinMP does not provide an '**`ompcc`**' command, a slightly different approach must be taken. To build 'lap-odinmp' (the OdinMP-compiled version of the laplace program), we would use the following rules instead:

```
odinmp_ccd/makefile:
        LIBS=$(LDLIBS) odinmp_prep --targetDirectory=odinmp_ccd
--output=../lap_odinmp lap.c second.c

lap-odinmp: odinmp_ccd/makefile
        cd odinmp_ccd; make
```

The first rule will prepare the OdinMP build directory ('odinmp_ccd'), with a makefile to create '../lap-odinmp' from the source files 'lap.c' and 'second.c', linking the .
The second rule will build lap-odinmp by changing to the OdinMP build directory and building the program there. Since this rule depends on the existance of the makefile in the OdinMP build directory, this will be created if it does not already exist.

Specifics on the Provided Programs

odinmp_prep

Synopsis

`odinmp_prep` [`--output=`*output*] [`--targetDirectory=`*targetDirectory*] [`--targetDebug=`(`always`|`runtime`|`never`)] [`--targetTiming=`(`always`|`runtime`|`never`)] *source-files*

Description

`odinmp_prep` prepares a project of C language source code files for OdinMP parallelization, by the following steps:

- creates the directory *targetDirectory*. If no *targetDirectory* is specified, it defaults to 'odinmp_ccd'.
- creates a makefile inside *targetDirectory* with a dependancy of each of the *source-files* to the corresponding file in the directory that odinmp_prep was called from.
- configures the `makefile` to call `OdinMP` with the `targetDebug` and `targetTiming` options as specified on the command line. Their respective default value is `never`.

- configures the `makefile` to build a resulting program called *output*. If *output* is not specified, it defaults to `a.out`.

OdinMP

Synopsis

`OdinMP [--targetDirectory=`*targetDirectory*`]`
`[--targetTiming=(always|runtime|never)] [--targetDebug=(always|runtime|never)]`
`[--cpp=`*cpp*`] [--cc=`*cc*`] [--numProcessors=`*numProcessors*`] [--numThreads=`*numThreads*`]`
source-files

Description

OdinMP performs the actual compilation from C with OpenMP directives, to C with pthreads library calls.
Files are read from the current directory and generated to *targetDirectory*. If *targetDirectory* is not specified, it defaults to 'odinmp_ccd'.
The 'targetTiming' option specifies whethet OdinMP/CCp will include timing code in the generated code, and whether timing will actually be performed. If *targetTiming* is 'never', no code is generated. If *targetTiming* is 'runtime', code is generated which will allow the resulting program to print timing information if called with the '--odinmp_timing' command line option. If *targetTiming* is 'always', the generated code will always measure and report timing information. The default value for *targetTiming* is 'never.
The 'targetDebug' option specifies whethet OdinMP/CCp will include debugging code in the generated code, and whether debugging messages will actually be printed. If *targetDebug* is 'never', no code is generated. If *targetDebug* is 'runtime', code is generated which will allow the resulting program to print debug information if called with the '--odinmp_debug' command line option. If *targetDebug* is 'always', the generated code will always print debugging information. The default value for *targetDebug* is 'never.
The '`--cpp=`*cpp*' option specifies an alternate C preprocessor to use for preprocessing the input C source files. The default is '*install-root*`/odinmp/bin/cppp`'.
The '`--cc=`*cc*' option selects an alternate C compiler to use for compiling the generated C source files. The default is '*install-root*`/odinmp/bin/ccc`'.

cppp

Synopsis

`cppp` [options] < input > output

Description

`cppp` reads un-preprocessed C source code from its standard input, preprocesses it, and prints the result to its standard output.
`cppp` is implemented as a Perl script which wraps around various different C preprocessors. By setting the variable '`$cpp`' at the top of the script, one of several underlying C preprocessors can be selected. The purpose of `cppp` is to insulate OdinMP from the particular semantics of how the underlying C preprocessor accepts and generates data.

The environment variable `ODINMP_CPP` can be used to override the contents of the `$cpp` variable specified in the script at runtime.

ccc

Synopsis

`ccc` [options] *source-files*

Description

`ccc` compiles the C source code specified into object files, or links the object files specified into an executable.
`ccc` is implemented as a Perl script which wraps around various different C compilers. By setting the variable '`$cc`' at the top of the script, one of several underlying C compilers can be selected.
The purpose of `ccc` is to insulate OdinMP from the particular semantics of how the underlying C compiler wants its arguments.
The environment variable `ODINMP_CC` can be used to override the contents of the `$cc` variable specified in the script at runtime.

Appendix E The OdinMP/CCp Source Code Distribution

This appendix reproduces verbatim the html document which describes how to unpack and compile OdinMP/CCp from its source code distribution.

The OdinMP/CCp Source Code

Requirements

To build OdinMP/CCp from source, you need:

- A Java Development Kit, Java version 1.1 or newer
- Perl version 5, available from The Perl Institute.
- JavaCC - the Java Compiler Compiler - available from MetaMata Inc.
- JTB - the Java Tree Builder for JavaCC - available from Purdue University.
- GNU make, avaliable from The Free Software Foundation and its mirror sites.
- A suitable directory in your Java class path
- And, of course, the OdinMP/CCp source archive itself, odinmp-1.0-src.tar.gz.

Unpacking the OdinMP/CCp Source Code Archive

The odinmp-1.0-src.tar.gz archive will, when unpacked, create a directory called 'odinmp', populated with the files necessary to build OdinMP/CCp.
OdinMP/CCp is implemented as a set of classes in the 'se.lth.dit.odinmp' package and its subpackages. This means that, in order to successfully compile the java code, the java source files must be in a directory '`se/lth/dit/odinmp`' beneath a directory that is in your Java Class Path, usually identified by the `CLASSPATH` environment variable. For instance, if your `CLASSPATH` is set to include a directory named 'java' in your home directory, you would create the directory *your-home-directory*`/java/se/lth/dit`, and then unpack the OdinMP/CCp source archive in this location - generating the directory *your-home-directory*`/java/se/lth/dit/odinmp`, which is exactly what the Java compiler expects.

Code Layout

These files are included in the distribution:

c.jtb	The Java Tree Builder / JavaCC grammar for C with OpenMP directives
OdinMP.java	The Java language source file for the OdinMP class, which implements the OdinMP main program
*ConstructDescriptor.java	The various OpenMP Construct descriptor classes
*ClauseHandler.java	Classes which handle clauses on constructs
Decl.java	Classes used to parse a part of a Declaration
Symbol.java	Class to hold information about a Symbol
SymbolTable.java	A Class to hold a table of Symbols
OmpSymbolAttributes.java	The class that handles symbols with OpenMP attributes (shared, private, reduction, ...)
IncrementalReader.java	Reads files in increments, used for generating code
Spacing.java	Class that helps OdinMP/CCp generate slightly less ugly code
SymbolFetcher.java	An interface between ClauseHandlers and the DataBuildingVisitor (see

	below)
OdinMP.in, odinmp_prep.in, cppp.in, ccc.in	Source files for the OdinMP, odinmp_pre, cppp and ccc programs
dodebug.pl, nodebug.pl	Perl scripts which comment out / in the print statements to debug the OdinMP/CCp classes
visitor/DataBuildingVisitor.java	The visitor that builds the internal data from the node tree generated by the parser
visitor/*	Supporting classes for the DataBuildingVisitor
makefile	The makefile (for GNU make) that builds the whole system
supporting/*	Supporting C language files used by OdinMP/CCp when generating code

Building OdinMP/CCp

To build OdinMP/CCp, stand in the unpacked source directory (*dir-in-classpath*`/se/lth/dit/odinmp`), make certain that you have all the necessary tools (as mentioned above) in your search path, and simply give the command '`make`' (or '`gmake`', if you have GNU make installed under that name). This will use JTB to create the syntax tree classes from c.jtb as well as create a tree-building grammar, jtb.out.jj. JavaCC will then work on that to generate the C Parser and supporting classes. Finally, all Java classes will be compiled together.

To install the resulting system, use '`make INSTALLDIR=`*install-root*`/odinmp install`'. This will install OdinMP/CCp in the install-root/odinmp directory, ready to use.

To create a gzipped tar archive of the installation package, use '`make package`'. This will generate the file '`odinmp-1.0.tar.gz`' in the OdinMP/CCp source code directory.

Appendix F References

1 The OpenMP ARB: OpenMP C and C++ Application Program Interface, Version 1.0 – October 1998
OpenMP Consortium Document Number 004–2229–001
Available on the WWW at <http://www.openmp.org/>

2 Brian W. Kernighan and Dennis M. Ritchie : The C Programming Language, Second Edition
Prentice Hall, Inc., 1988, ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback)

3 The Open Group: The Single UNIX Specification, Version 2 - 6 Vol Set for UNIX 98
Open Group Publication Set Document Number T912
Also available on the WWW at <http://www.opengroup.com/unix/>

4 David E Culler, Jaswinder Pal Singh, with Anoop Gupta: Parallel computer architecture: a hardware/software approach
Morgan Kaufman Publishers, Inc., 1999, ISBN 1-55860-343-3

5 Kai Hwang, Zhiwei Xu: Scalable parallel computing: technology, architecture, programming
WCB/McGraw-Hill, 1998

6 The Message Passing Interface Forum: MPI: A Message-Passing Interface Standard
Available on the WWW at <http://www.mpi-forum.org/>

7 Sriram Sankar, Sreenivasa Viswanadha, Rob Duncan, Juei Chang: JavaCC, The Java Compiler Compiler
Available on the WWW at <http://www.metamata.com/JavaCC/>

8 Dr. Jens Palsberg, Kevin Tao, Wanjun Wang: The Java Tree Builder
Available on the WWW at <http://www.cs.purdue.edu/jtb/>

9 Bill Magro: A simple molecular dynamics simulation, using the velocity Verlet time integration scheme
Kuck and Associates, Inc. (KAI), 1998
Available on the WWW at <http://www.openmp.org/samples/md.f>