CPDS: OpenMP Hands-On

E. Ayguadé and D. Jiménez

November 22, 2012

Index

1

Introduction

This document is intended to guide you through a set of codes that we provide to introduce OpenMP, the standard the facto programming model for shared memory architectures. This session does not have deliverables but we recommend that you follow this document carefully in order to understand how OpenMP works. You will need:

- The set of slides for *Short tutorial on OpenMP* available through the Wiki.
- The set of files in `/home/cpds92/cpds92000/OMP_handson.tar.gz`.

First you will look at a set of very simple examples that introduce the main components of the OpenMP programming model (chapter 2 in this document). We recommend that you look at the tutorial slides in order to fully understand them. After that, you will parallelize the computation of number pi, going through a set of versions (some of them not correct) that will further exemplify the use of parallel regions, worksharing constructs and tasking (chapter 3 in this document).

2

OpenMP tutorial examples

2.1 OpenMP basics

1. Get into the directory called `basics`. The examples are ordered. Look at each code and try to answer the questions that are included in the source code; later you can compile each code and run to check your answers.
2. Consult `Part I: OpenMP Basics` of the tutorial slides if necessary.

2.2 Loop parallelism

1. Get into the directory `worksharing`. Look at the *1.for.c* and *2.collapse.c* codes and predict the output before running the program; later you can compile the codes and run to check your answers.
2. Consult `Part II: Loop Parallelism in OpenMP` if necessary.

2.3 Task parallelism

1. Get into the directory `tasks`. Look at the serial version in *linked_serial.c*. It implements a linked list which is traversed and some computation is done for every node of the list in the function called `processwork`. The computation consists on generating the ith number of the Fibonacci series where i is the *data* value of the node. Run the code and see how it works.
2. The file *linked_v1.omp.c* presents a parallel version of the same code using the **task** construct. This code is not correct, try to run it and see what happens. The problem with tasks is that they need to capture the value of the data they need when they are created. In this case this was not done and when the task was accessing p it was probably at the end of the list so it was pointing to $NULL$.
3. Next version *linked_v2.omp.c* uses the **firstprivate** clause to capture p at the time the task is created. Run and check the result.
4. Consult `Part III: Task Parallelism in OpenMP` if necessary.

3

Parallelizing the computation of Pi with OpenMP

3.1 Computing number Pi

As an example we will use a program that computes the number pi by solving the equation in Figure 3.1. The equation can be solved by computing the area defined by the function, which at its turn it can be approximated by dividing the area into small rectangles and adding up its area.

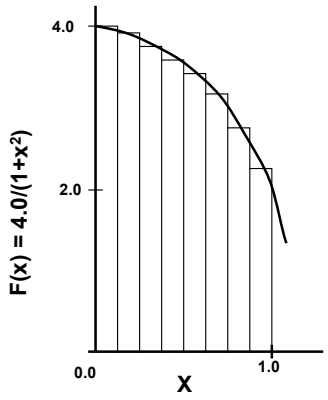
Figure 3.2 shows the sequential code for pi computation: To distribute the work for the parallel version each processor will be responsible for computing some rectangles (in other words, to execute some `i` iterations). It should be guaranteed that all processors have computed their `sum` before combining it into the final value.

3.2 Parallelization with OpenMP

In this section we will parallelize the sequential computiation of Pi described before. We will proceed through a set of versions, some of them incorrect, to finally get to the final version. To compile any of the codes[1] just type "`make pi_omp_vx.omp`", being `x` the version number.

1. In a first attempt to parallelize the sequential code we have introduced a `parallel` construct. The `parallel` construct creates the team of threads or reuses them if they have been created before. In OpenMP all variables are shared by default and usually some of them would need to be privatized. The code is in *pi_omp_v1.c*. Run it with small number of iterations (i.e. 10). Notice which iterations are executed by each thread and how many. This code is NOT correct, one of the problems is that the threads replicate the work and the loop control variable is shared.

[1] All versions are inside the `pi` directory.



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Figure 3.1: Pi computation

```
static long num_steps = 100000;
void main ()
{
        int i;
        double x, pi, step, sum = 0.0;

        step = 1.0/(double) num_steps;

        for (i=1;i<= num_steps; i++) {
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Figure 3.2: Serial code for Pi

2. Next version *pi_omp_v2.c* has added the `private` clause for variables i and x. Now that i is private we can see that all threads execute all iterations of the loop.

3. Next version *pi_omp_v3.c* uses the runtime call `omp_set_num_threads()` and changes the for loop in order to distribute the iterations of the loop among the threads. Run the code with small number of iterations and check what iterations are executed by each thread.

4. However the previous version has a race condition, threads need to be synchronized. In the previous version threads only synchronize at the end of the parallel region because there is an implicit barrier. Next version *pi_omp_v4.c* uses the `critical` construct to provide a region of mutual exclusion where only one thread can be working at any given time. An alternative implementation could use the lock mechanism provided by the OpenMP runtime. And finally, this is a correct parallel implementation of the pi program.

5. The file *pi_omp_v5.c* uses the `atomic` construct to guarantee mutual exclusion. The *atomic* construct provides an special mechanism of mutual exclusion for simple read and update operations, which is more efficient that the `critical`. Run it and compare the time with the previous version.

6. An alternative solution is used in *pi_omp_v6.c*: the `reduction` clause. Reduction is a very common pattern where all threads accumulate values into a single variable. The compiler creates a private copy of each variable and at the end of the region, the compiler ensures that the shared variable is properly updated with the partial values of each thread, using the specified operator. Check the code and run it.

7. Next we will use the `for` construct to distribute the iterations of the loop among the threads of the team. This is the *pi_omp_v7.c* version. Run it and notice which iterations are assigned to each thread.

8. The `schedule` clause determines which iterations are executed by each thread. There are three different options as schedule: (i) `static` (ii) `dynamic` and (iii) `guided`. An explanation of each schedule and his options can be found in the slides from page 52 to 54. Review how each schedule works and use the provided examples to try out the different schedules: *pi_omp_v7.c* and *pi_omp_v8.c* for static; *pi_omp_v9.c* and *pi_omp_v10.c* for dynamic; and *pi_omp_v11.c* and *pi_omp_v12.c* for guided.

9. Version *pi_omp_v13.c* introduces the use of `omp_get_wtime` runtime call to measure wall clock execution time.

10. Versions *pi_omp_v14.c* and *pi_omp_v15.c* exemplify the use of the `nowait` clause. When a worksharing has a `nowait` clause then the implicit barrier at the end of the loop is removed. Imagine that we want to compute the time each thread spends in the computation of pi. We have changed the scheduler to `(static,num_steps-1)` to make it unbalanced. Check and run the example codes (with and without the `nowait` clause). The use of the `nowait` allows us to compute the correct time for each thread.

11. Version *pi_omp_v16.c* exemplifies the use of the `single` construct. We have modified the code and move the instruction `pi=step*sum` inside the parallel region. The problem is that all threads will execute it. To make sure that only one thread of the team will execute the structured block we must use the `single` construct. Run the code, is it correct?

12. Unfortunately the result is not correct because there is still the `nowait` clause and the implicit barrier at the end of the loop has been removed. To fix the problem and still be able to compute the time of both threads we can use the `barrier` construct. Check the code in *pi_omp_v17.c*

13. Finally we will use tasking to parallelize the code. The `task` construct provides a way of creating a new explicit task. When the `task` construct is encountered a task is created that will later run in parallel. The values of the variables which are firstprivate type are captured at creation time. To exemplify the use of the **task** construct we have modified the pi program: *pi_omp_v18.c*. For that purpose we have removed the `for` construct and added the `task` construct inside the loop so that each iteration of the loop will be a task. We also need to add the `atomic` again. Notice that in the case of the pi program, using the for construct is the more effective way of parallelization, however remember that if the iterations of the loop cannot be counted you will not be able to use the for construct, in this case tasks are a good option. Look at the code, run it and check which iterations are executed by each thread.

14. The problem with the previous code is that threads are replicating the work, we fix it using the `single` construct so that only one thread generates the tasks: *pi_omp_v19.c*

The following table summarizes all the codes used during the process. Changes accumulate from one version to the following. In order to compile a program use `"make version_name.omp"`. For instance, `"make pi_omp_v1.omp"` to compile that version.

Code	Description of Changes	Correct?
v1.c	Added parallel construct and use of `omp_get_thread_num()`	no
v2.c	Added private for variables `x` and `i`	no
v3.c	Added `omp_set_num_threads()` and interleaved in the loop	no
v4.c	Critical in sum	yes
v5.c	Atomic in sum	yes
v6.c	Reduction on sum	yes
v7.c	Add omp for (schedule by default in the omp for: static)	yes
v8.c	Example of schedule static,1 in omp for	yes
v9.c	Example of schedule dynamic in omp for	yes
v10.c	Example of schedule dynamic,2 in omp for	yes
v11.c	Example of schedule guided in omp for	yes
v12.c	Example of schedule guided,4 in omp for	yes
v13.c	Introduces the use of `omp_get_wtime`	yes
v14.c	Timing omp for, with static,num_steps-1: threads same time	yes
v15.c	Timing omp for, with static,num_steps-1 and nowait	yes
v16.c	Introduces single construct	no
v17.c	Introduces barrier construct	yes
v18.c	Uses the task construct	no
v19.c	Added the single construct	yes