Assignment 1

Patrick Indri

March 31, 2019

1 Introduction

The aim of this assignment is to solve two exercises concerning OpenMP programming. The work is organized into two folders (`/Ex_1` and `/Ex_2`), one for each exercise.

2 Exercise 1

The first exercises asks to approximate π using numerical integration. In particular, π can be approximated using the mid-point quadrature rule:

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx \approx 4 \sum_{i=0}^{N-1} h f(x_{i+1/2})$$

where N is the number of quadrature points and $h = 1/N$ is the distance between two consecutive points.

`pi.c` implements both a serial and parallel version of the approximation. The `makefile` provided compiles the code into an executable which prints the approximation of π and the execution time for both the serial and parallel computation. Running `make benchmark`, instead, compiles the code with a simplified output, where only the execution times are printed: this choice simplifies the subsequent plotting of the data.
`run.sh` is the script I used to run the code on Ulysses cluster. For each number of

processing nodes it performs 4 executions. The results are shown in the two following plots.

Fig. 2.1: Strong scaling, $N = 10^8$, mean of 4 runs.

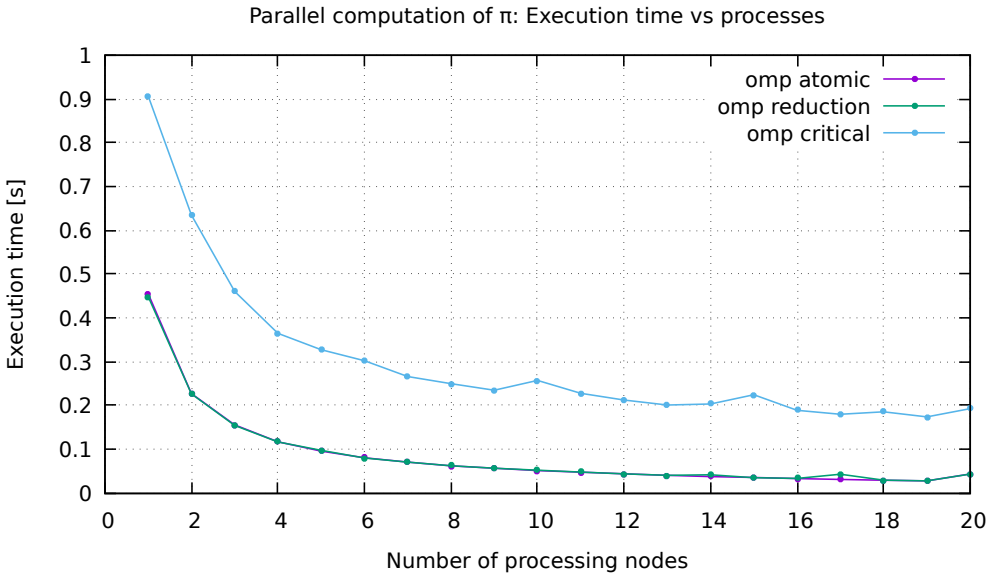

Fig. 2.2: Speedup, $N = 10^8$, mean of 4 runs. The dashed lined represents linear speedup.

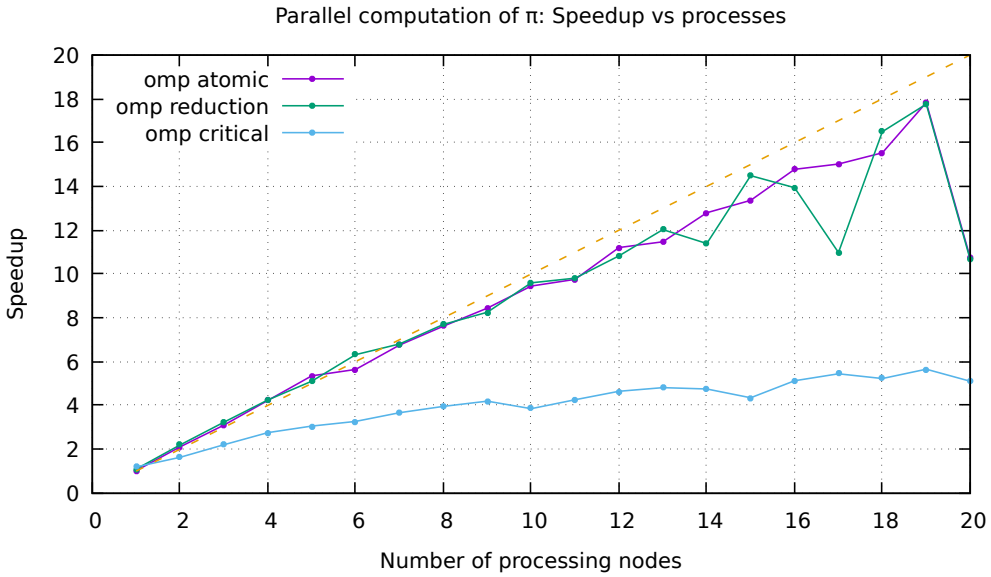

Fig. 2.1 and Fig. 2.2 have been obtained using $N = 10^8$ quadrature points. Both plots compare the different methods of protection for the summation variable. As expected `omp critical` offers the worst performance, resulting in longer execution times and worse speedup. `omp reduction` and `omp atomic`, on the other hand, perform quite similarly, almost achieving linear speedup. In particular, `omp atomic` shows a more stable speedup. This behaviour has no clear explanation, and can probably be attributed to random fluctuations (increasing the number of runs could probably stabilise the results). It is noteworthy that these last two methods to deal with critical sections offer superlinear speedup for a small number of processors.

3 Exercise 2

The second exercise deals with OpenMP loop schedules. The aim is to visualise how different schedules distribute work among the available processors.

`loop_schedule.c` implements the solved exercise. The included `makefile` should be use to compile the code. The implementation allocates an array of dimension N which is subsequently filled by the available: each thread will fill part of the array, following the adopted schedule.
The function `print_usage` (already provided) has been used to visualize the result. As required, I compared the behaviour of `static` and `dynamic` schedules with different chunk sizes using only one `omp parallel` region (`omp single` has been exploited so that only one processor calls `print_usage`). I run the code on Ulysses cluster using the script `run.sh`.

The results obtained are now briefly commented. For the sake of visualisation, consider the case $N = 40$ and 4 threads:

- `static` equally distributes the work among the threads. In particular, each thread will be assigned a chunk of $N/n_{threads}$, following the order of the threads;

```
static
0: **********
1:            **********
2:                       **********
3:                                  **********
```

- `static, chunk size = 1` equally distributes the work among the threads. In particular, each thread will be assigned a chunk of size 1, following the order of the threads;

```
static, chunk size = 1
0: *    *    *    *    *    *    *    *    *    *
1:  *    *    *    *    *    *    *    *    *    *
2:   *    *    *    *    *    *    *    *    *    *
3:    *    *    *    *    *    *    *    *    *    *
```

- `static, chunk size = 5` equally distributes the work among the threads. In particular, each thread will be assigned a chunk of size 5, following the order of the threads;

```
static, chunk size = 5
0: *****                *****
1:      *****                *****
2:           *****                *****
3:                *****                *****
```

- `dynamic` does not necessarily distributes the work equally among the threads. The work distribution is determined at run-time, using no constraints on chunk size;

```
dynamic
0: *    ******  * * *  * ***    *    *     *
1:   *        **    *  *        *         *    *
2:    *          *    *  *    *      *    *    *
3:  *                         *  **  **    *
```

- `dynamic, chunk size = 1` does not necessarily distributes the work equally among the threads. The work distribution is determined at run-time, using chunk of size 1;

```
dynamic, chunk size = 1
0:    *  *     * * *   *     * **  *     **
1: * *  *   **      * * *     *      *  *
2:  *  *   *    *    *    *       *  * *
3:        *   *   *      * **     *    *
```

- `dynamic, chunk size = 5` does not necessarily distributes the work equally among the threads. The work distribution is determined at run-time, using chunk of size 5;

```
dynamic, chunk size = 5
0: *****           *****
1:            *****           *****      *****
2:                                  *****
3:       *****           *****
```

In this particular case using a `static` schedule guarantees a better workload balancing. It should be underlined that this is not always the case: a `static` schedule with a chunk size of 20 would result in a highly unbalanced workload distribution. Moreover, if each iteration takes different time to be completed, work unbalance may arise as well. Consider the case where the iteration time (in our case, the time to write an element of the array) increases with the iteration number: `static` will equally distribute iterations, but actual workload will be unequally distributed, the last threads taking much longer to execute their iterations. If this is the case, then a `dynamic` scheduling would perform better.
Another reason to consider when choosing between `static` and `dynamic` scheduling is overhead: sharing workload is done at compile time with `static` scheduling and at run time with `dynamic` scheduling (which means additional overhead for communication).