Final Project Report of

Parallel Programming Fundamentals

Matteo Barbetti

21st July 2022

1 Introduction

This document reports the methodology followed and the results obtained for the final project of "Parallel Programming Fundamentals", a PhD course of the University of Siena given by Roberto Giorgi and Marco Procaccini, and attended from 23rd to 29th May 2022.

The aim of the project is to quantify the *parallelism* achieved by a program using the OpenMP APIs [1] to speed up the computation of the Mandelbrot set [2]. The program used for this study is written in C and corresponds to a slightly modified version of the `omp-mandelbrot.c` script provided within the course materials [3].

The elapsed time of the program has been measured for different types of *scheduling* and *partition sizes*, and its performance has been tested in *weak* and *strong scaling* conditions, as better described in Section 2. The results of the scheduling study are reported in Section 3.1, while the speedup achieved with weak and strong scaling conditions are shown in Section 3.2. In Section 4, some general remarks are discussed, and an estimation of the parallelism of the program is given.

2 Methodology

The elements of the Mandelbrot set are computed by the recursive function `iterate` that follows:

```
int iterate( float cx, float cy )
{
  float x = 0.0f, y = 0.0f, xnew, ynew;
  int it;
  for ( it = 0; (it < MAXIT) && (x*x + y*y <= 2.0*2.0); it++ ) {
    xnew = x*x - y*y + cx;
    ynew = 2.0*x*y + cy;
    x = xnew;
    y = ynew;
  }
  return it;
}
```

Calling such function has a highly variable computational cost that depends on the number of loop iterations that, in turn, depends on the values of `cx` and `cy`, and on the maximum number of iterations allowed (`MAXIT`). In `omp-mandelbrot.c`, the `iterate` function is called within a two-level nested loop that computes the Mandelbrot set for a 2-D array having 768 rows (`y_size`) and 1024 columns (`x_size`), while the maximum number of iterations is set to 10000.

For the studies of this project, a new set of scripts[1] has been prepared, starting from a new source code, named `my-mandelbrot.c`. The main difference with respect to `omp-mandelbrot.c` is the memory allocation to store the elements of the Mandelbrot set, in order to export them for graphical visualization (see Figure 1).

The program is parallelized using a set of OpenMP APIs that wraps the nested loop as shown in the following:

[1] All scripts, data and images produced for this project are available at `mbarbetti/ppf-omp-project`.

```
  // [...]
#pragma omp parallel for private(x) schedule(runtime)
  for ( y = 0; y < y_size; y++ ) {
    for ( x = 0; x < x_size; x++ ) {
      const double re = x_min + (x_max - x_min) * (float)(x) / (x_size - 1);
      const double im = y_max - (y_max - y_min) * (float)(y) / (y_size - 1);
      const int it = iterate(re, im);   // highly variable work
#pragma omp critical
    if ( it < MAXIT ) {
      matrix[y*y_size + x] = it;   // saved for visualization
      }
    }
  }
  // [...]
```

The use of the `omp for` directive and of a private scope for the `x` variable means that only the inner loop is parallelized and its work distributed across the threads assigned the executing program by the `omp parallel` directive. The time needed for completing the nested loop is measured (namely, the *elapsed time*) and used performance studies.

In addition to the single-loop parallelization

```
  // [...]
#pragma omp parallel for collapse(2) schedule(runtime)
  // [...]
```

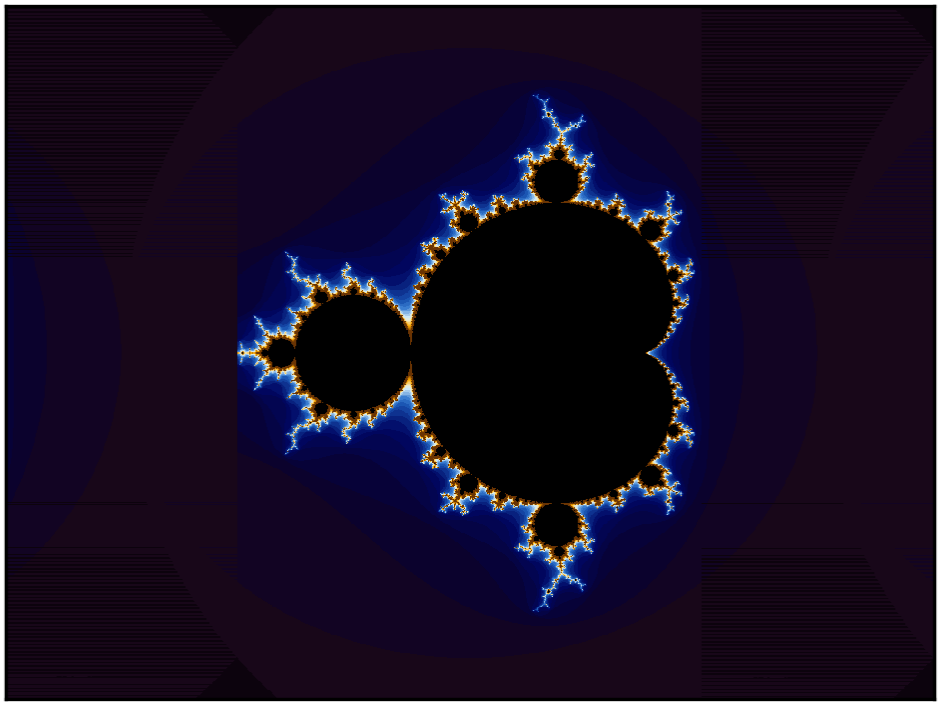

Figure 1:

2.1 Scheduling study

2.2 Scaling study

3 Results

3.1 Scheduling performance

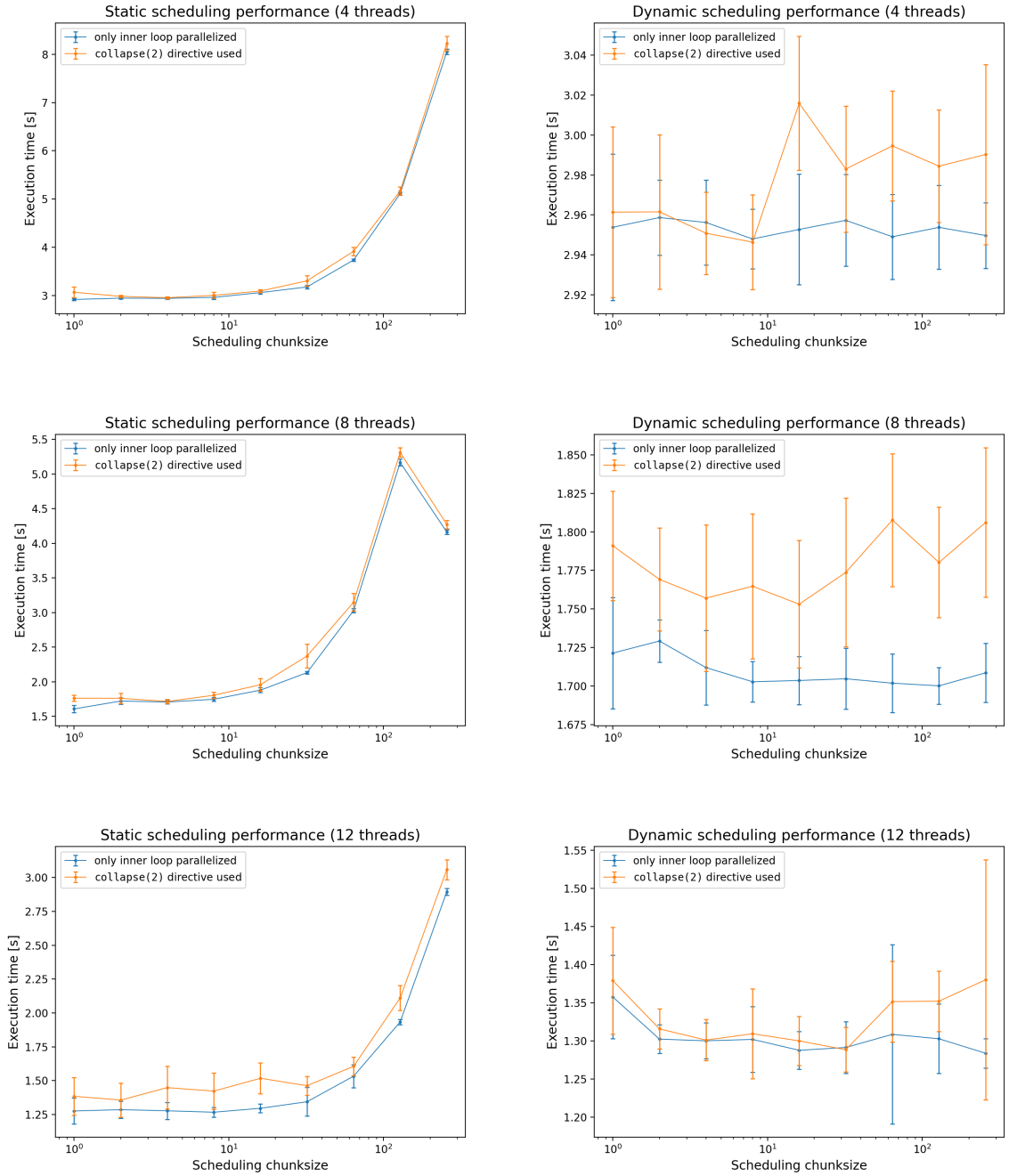

Figure 2:

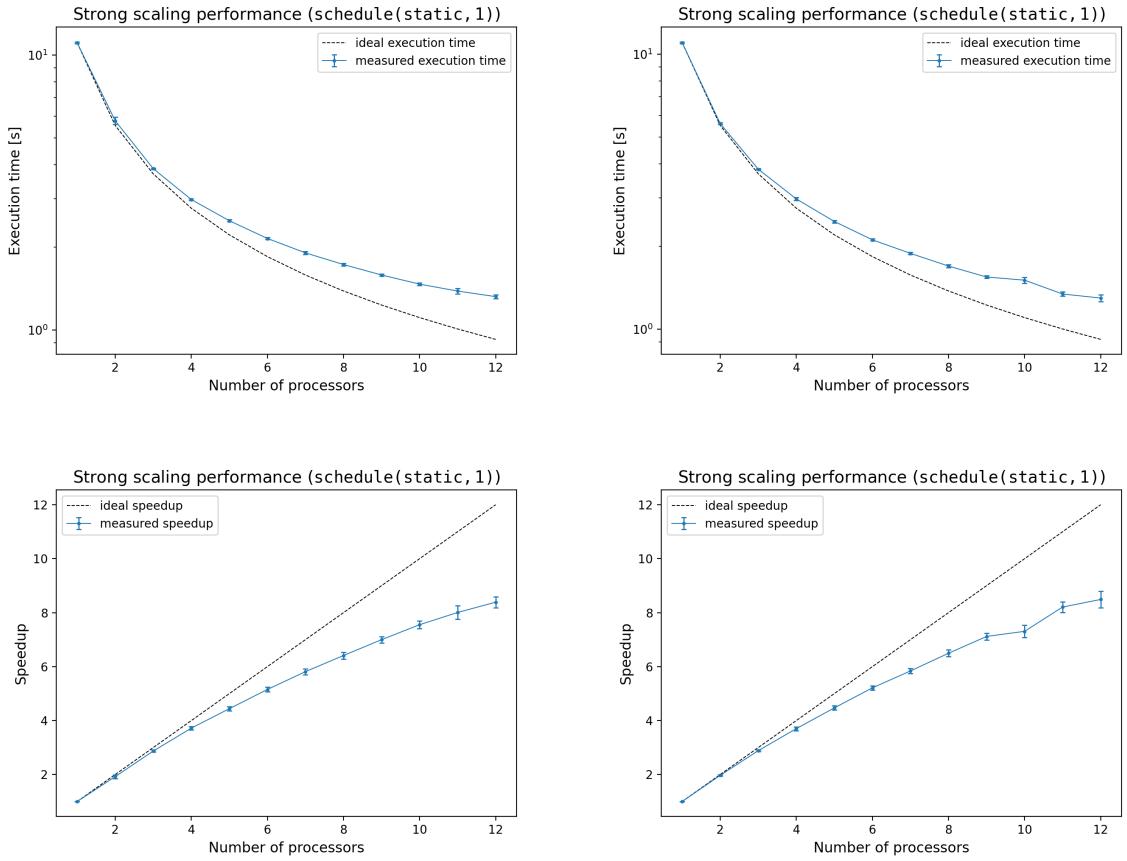

Figure 3:

3.2 Scaling performance

4 Conclusion

References

[1] https://www.openmp.org/.

[2] https://en.wikipedia.org/wiki/Mandelbrot_set.

[3] https://classroom.google.com/u/1/c/NTEwNTQ3NTIyMzcx.

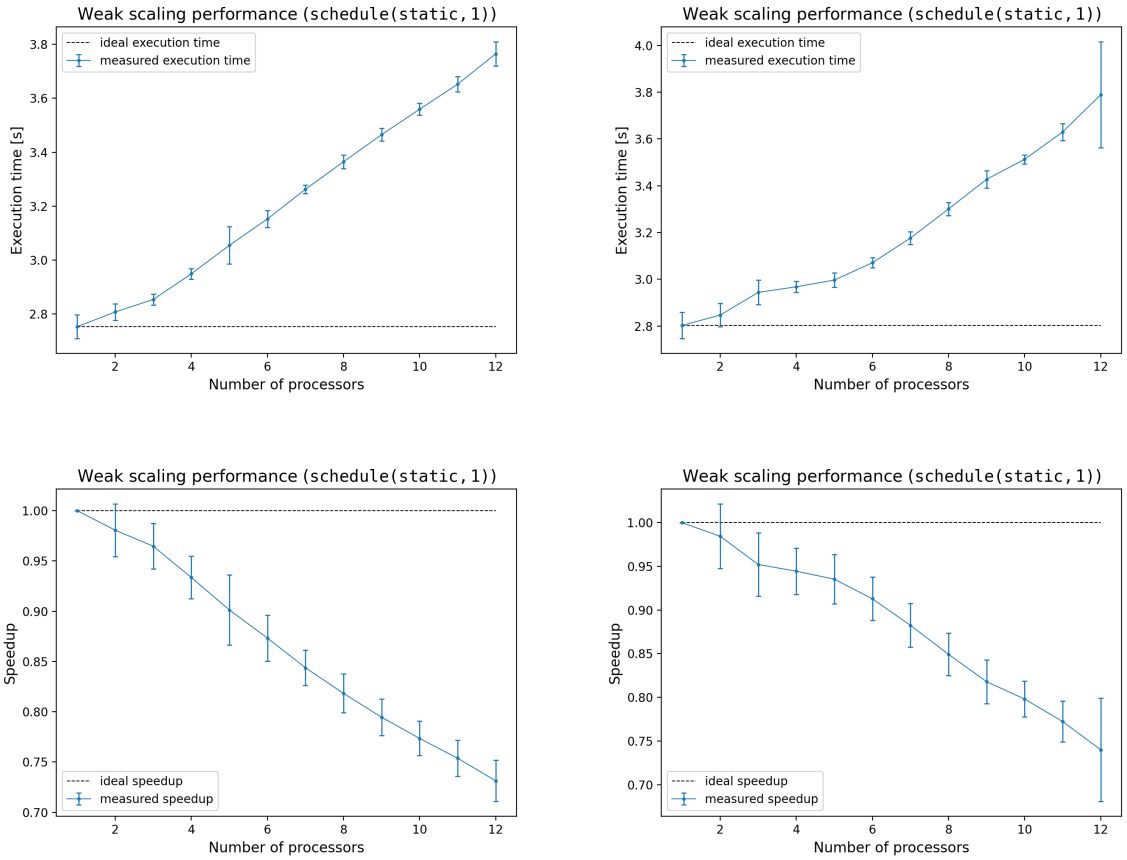
Weak scaling performance (schedule(static,1))
ideal execution time
measured execution time
Execution time [s]
Number of processors
Weak scaling performance (schedule(static,1))
ideal execution time
measured execution time
Execution time [s]
Number of processors
Weak scaling performance (schedule(static,1))
ideal speedup
measured speedup
Speedup
Number of processors
Weak scaling performance (schedule(static,1))
ideal speedup
measured speedup
Speedup
Number of processors

Figure 4: