Final Project Report of

Parallel Programming Fundamentals

Matteo Barbetti

22nd 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] on a single machine. 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 in the executing program by the `omp parallel` directive. The time needed for completing the nested loop is measured (namely, the *elapsed time*) and used for performance studies.

In addition to the inner loop parallelization, a second script using the `collapse(2)` clause has been prepared. It makes the `x` and `y` variables private by default and allows to distribute the work of the whole nested loop across the various threads reached by the `omp parallel` directive.

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

In the following parts of this document, the elapsed times needed to compute the Mandelbrot set parallelizing only the inner loop (with `my-mandelbrot.c`) or the whole nested loop (with `my-mandelbrot-collapse.c`) are reported for different types of scheduling and partition sizes. Moreover, the performance in weak and strong scaling conditions are discussed for both the parallelization strategies.

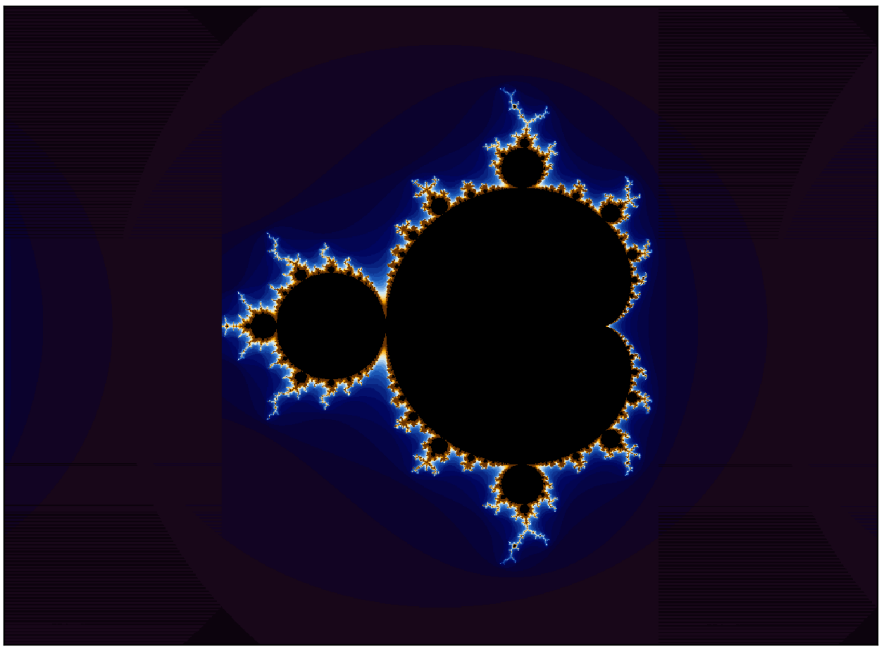

Figure 1: The Mandelbrot set (black) within a continuously colored environment.

2.1 Machine

All the performance studies have been carried on a single machine equipped with an Intel Core i7-9750H @ 2.60 GHz and 6 CPU cores. The Mandelbrot programs (and each bash script) have been executed within the Docker image `ppf-shm05:latest` [3] through an Ubuntu 20.04 instance powered by WSL 2 [4] on Windows 10.

2.2 Scheduling study

The performance of both the Mandelbrot programs has been studied using different scheduling strategies. In particular, the scheduling types investigated have been the `static` and `dynamic` ones, while the scheduling chunksize has been varied from 1 to 256.

Two bash scripts have been prepared for this purpose, named `size-partition.sh` and `size-partition-collapse.sh`. The latters, taken the number of threads to use and the scheduling type at run-time, pass the information to the respective executable file through the environment variables `OMP_NUM_THREADS` and `OMP_SCHEDULE`. The programs are then executed within a loop that assign various scheduling chunksize values at run-time. In order to obtain results more robust, each measurement of the elapsed time for a specific configuration is taken 20 times. The test has been repeated passing 2, 4, 6, 8, 10 and 12 threads to the bash scripts to study the behavior of the various scheduling strategies by varying the available computing processors.

2.3 Weak and strong scaling study

Study the performance in weak scaling conditions corresponds to expand the input data size so that the work performed by each threads remains constant: $T_P = T_1$. Since the presence of a two-level nested loop, $O(m \times n)$, in order to satisfy these requirements the size of the Mandelbrot matrix (`y_size`, `x_size`) should increase by a factor $\sqrt{P}$ with P the number of processors available. For this study, since the Mandelbrot matrix sizes are increased up to a factor 3.5 ($\sqrt{12} \simeq 3.5$), the initial values of `x_size` and `y_size` are set to 512 and 384 respectively.

On the other hand, in strong scaling the input data size is kept constant so that the total work is constant. Then, what expected is that the work performed by each thread linearly decrease with the increasing of the number of processors available: $T_P = T_1/P$. For this study, the initial values of `x_size` and `y_size` are kept constant to 1024 and 768 respectively.

Four bash scripts have been prepared for these studies, two based on the `my-mandelbrot.c` program named `weak-scaling.sh` and `strong-scaling.sh`, and two based on the `collapse` variant. All the bash scripts take the scheduling type and chunksize at run-time, and pass them to the respective executable file through `OMP_SCHEDULE`. The programs are then executed within a loop that assign a variable number of processors to `OMP_NUM_THREADS`. Finally, in order to obtain results more robust, each measurement of the elapsed time for a specific configuration is taken 20 times. After a simple statistical treatment, the measurements are combined to derive the speedup values.

3 Results

3.1 Scheduling performance

The Figure 2 shows the time needed to compute the Mandelbrot set with respect to the partition size in `static` (left plots) and `dynamic` (right plots) scheduling conditions. Looking at the plots reported on the left column is clear that greater values of the scheduling chunksize leads to an increasing of the overhead. This is an expected behavior if we look at the graphical representation of the Mandelbrot set reported in Figure 1: since the points with the highest computational cost are adjacent (the light brown ones), taking a large value for the chunksize corresponds to a high probability of assigning blocks of expensive computations to single threads, instead of

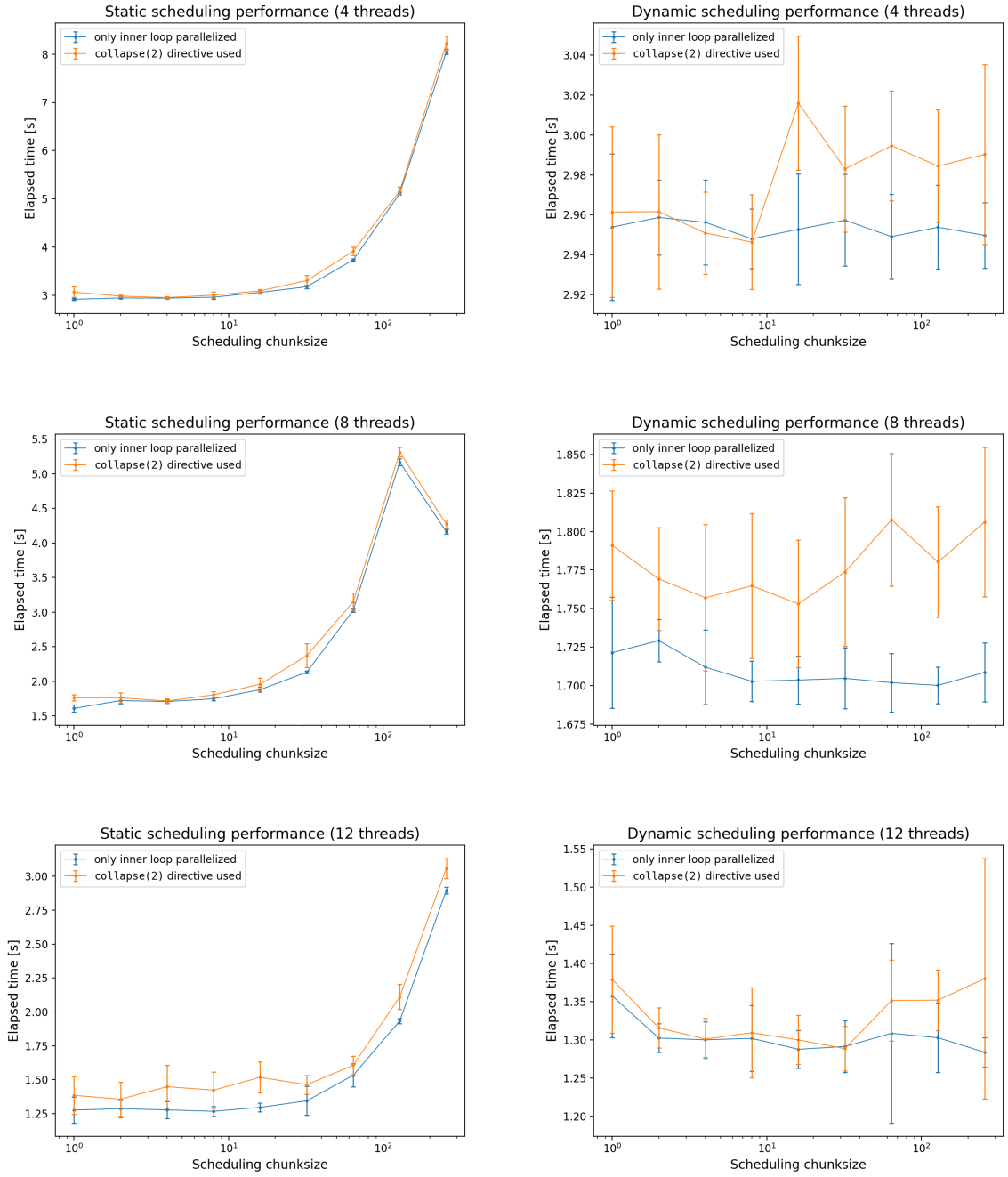

Figure 2: Performance curves of programs parallelized to compute the Mandelbrot set using 4, 8 and 12 threads. The left column reports the effect of enabling the `static` scheduling with different values of the chunksize in the elapsed times of a program with only the inner loop sped up (blue curves) and a program with the whole nested loop parallelized (orange curves). On the right column the same curves are reported using the `dynamic` scheduling.

distributing the workload as it happens with low values of the chunksize. On the contrary, the `dynamic` scheduling seems to be not affected by this drawback: a possible explanation is that the dynamic assignment of the workload allows to the quicker threads to handle the works more expensive, partially absorbing the overhead reported in `static` scheduling. However, I believe this isn't enough to describe what shown in the plots on the right column.

Another interesting results to derive from the plots in Figure 2 is a comparison in performance terms of the program with only the inner loop parallelized against the one that parallelizes the whole nested loop. Even if in most of the points the elapsed times seem to be consistent, the `collapse`-based program shows performance slightly worse than its counterpart. Again, we are looking at a non-completely understood behavior that maybe can be described like a sort of edge-phenomenon: in fact, the `collapse(2)` clause allows to unroll the nested loop, leading the last iterations of the outer loop to be adjacent to the first ones of the next inner loop. Since the region near the edges almost doesn't require any computation (see the colors of the Figure 1), it can lead to a greater probability of having the *same* processors performing the heaviest work, for both `static` and `dynamic` scheduling.

3.2 Weak and strong scaling performance

As highlighted by the plots in Figure 3 representing the performance of the weak scaling conditions, the parallelization of both the Mandelbrot programs shows a behavior quite different from the ideal one. In fact, looking at the elapsed times on the first row, we can observe a *linear* trend that deviates from the expected one by $T_P \sim T_1 + \varepsilon \cdot P$, with $\varepsilon \simeq 0.08$ s (empirically evaluated). The same phenomenon can be appreciated on the speedup plot (second row of Figure 3) where the negative trend can be described by $S_P \sim 1 - \varepsilon \cdot P/T_P$. Here, the ε/T_P term can be read as

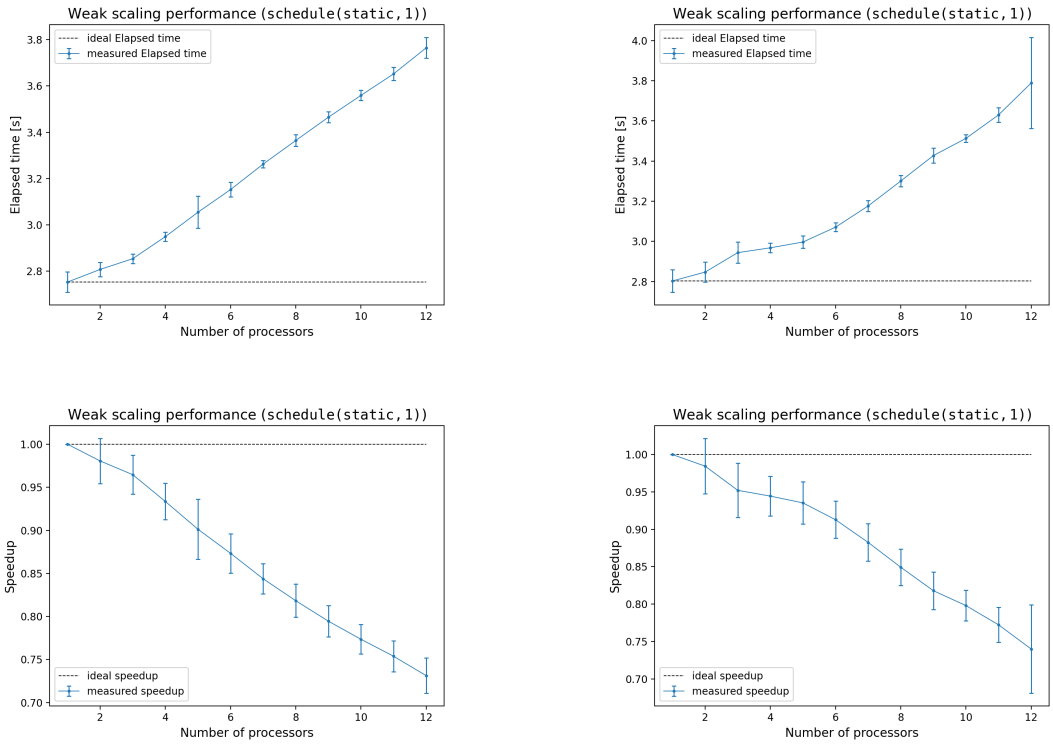

Figure 3: Performance plots (elapsed time and speedup) resulting from the *weak scaling* conditions for a program with only the inner loop sped up (left column) and a program with the whole nested loop parallelized (right column). Both the programs are executed with the scheduling strategy fixed to `schedule(static,1)`.

the percentage drop in performance (with respect to the speedup) brought by each thread, that in this case amounts to about 2%.

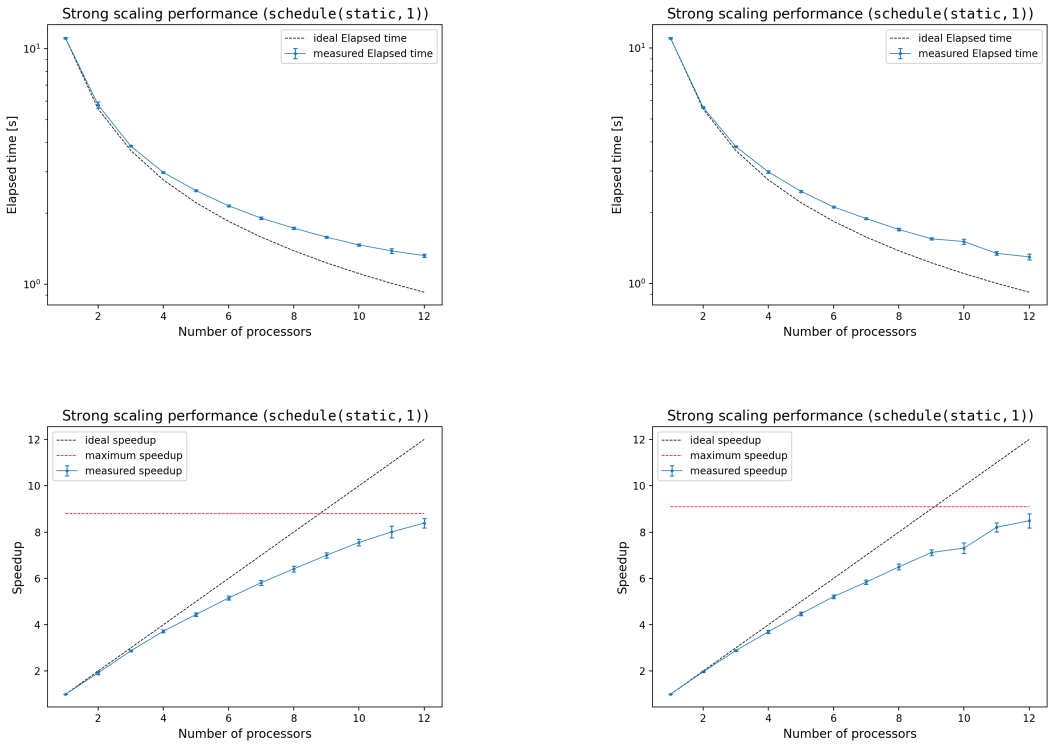

Figure 4: Performance plots (elapsed time and speedup) resulting from the *strong scaling* conditions for a program with only the inner loop sped up (left column) and a program with the whole nested loop parallelized (right column). Both the programs are executed with the scheduling strategy fixed to `schedule(static,1)`.

Also the plots reported in Figure 4 representing the performance with the strong scaling conditions show a significant deviance from the ideal behavior, namely $T_P = T_1/P$ and $S_P = P$. The latter plots confirm what derived from the discussion of weak scaling, namely that the computing threads seem to perform slightly worse than expected, accumulating an overhead that becomes more and more significant with the increasing of the involved threads. This is clearly in accordance with the *Work Law*. Moreover, relying on the *Span Law*, the plots on the second row of Figure 4 can be used to give an estimate, or better saying a lower bound of the parallelism intrinsic to the Mandelbrot program.

4 Conclusion

Some timing performance studies on programs to compute the Mandelbrot set have been carried out. Two programs based on OpenMP have been prepared: the first program designed to speed up only the inner loop of the nested loop used to fill the 1024×768 matrix, while the second one slightly modified to parallelized the whole nested loop through the `collapse(2)` clause.

Different scheduling strategies have been investigated, showing behaviors significantly inconsistent between the `static` and `dynamic` scheduling, especially for what concerns the overhead treatment. In particular, the results obtained for the `dynamic` scheduling varying the chunksize values are still not completely understood, and should require additional investigations.

Selected the best performing scheduling strategy, namely `schedule(static,1)`, studies on weak and strong scaling conditions have been carried out. For both the cases, the machine used

for the tests has shown performance significantly inconsistent with the ideal one. This is clearly in accordance with the Work Law that has occurred with a small amount of overhead brought by each thread, phenomenon well represented by the plots in Figure 3. We land to the same conclusion looking at the plots in Figure 4 that also contains an additional information. In particular, from the Span Law we know that the speedup achieved using various threads cannot overcome the maximum speedup, named *parallelism*. We can then used the trend of speedup reported on the second row of Figure 4 to extrapolate an estimate[2] of the parallelism: $S_{\max} \gtrsim 9$.

References

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

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

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

[4] `https://docs.microsoft.com/en-us/windows/wsl/`.

[2] Actually, better saying, the result reported is a lower bound and not an estimate of the parallelism.