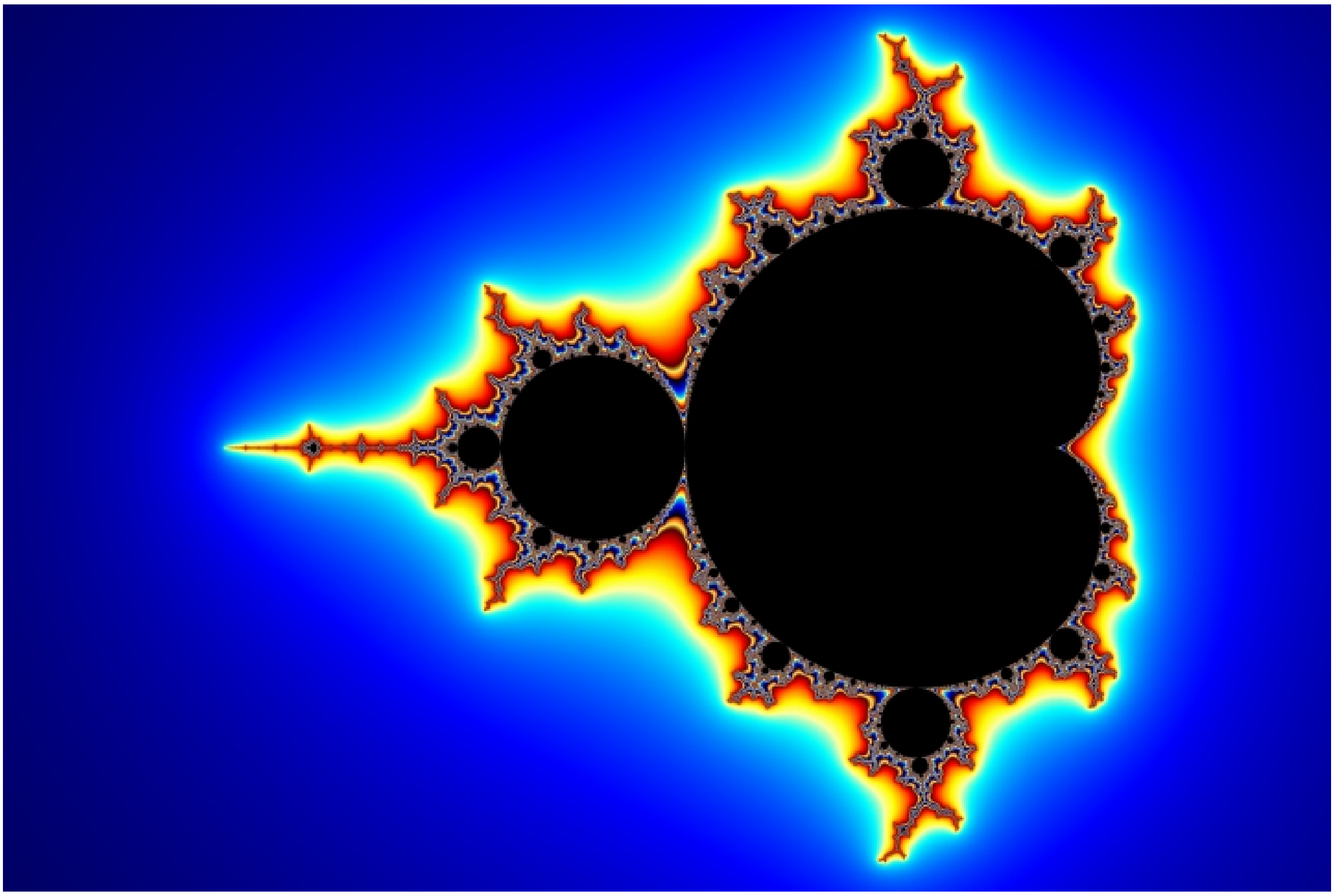
0,99562363

Lab 2

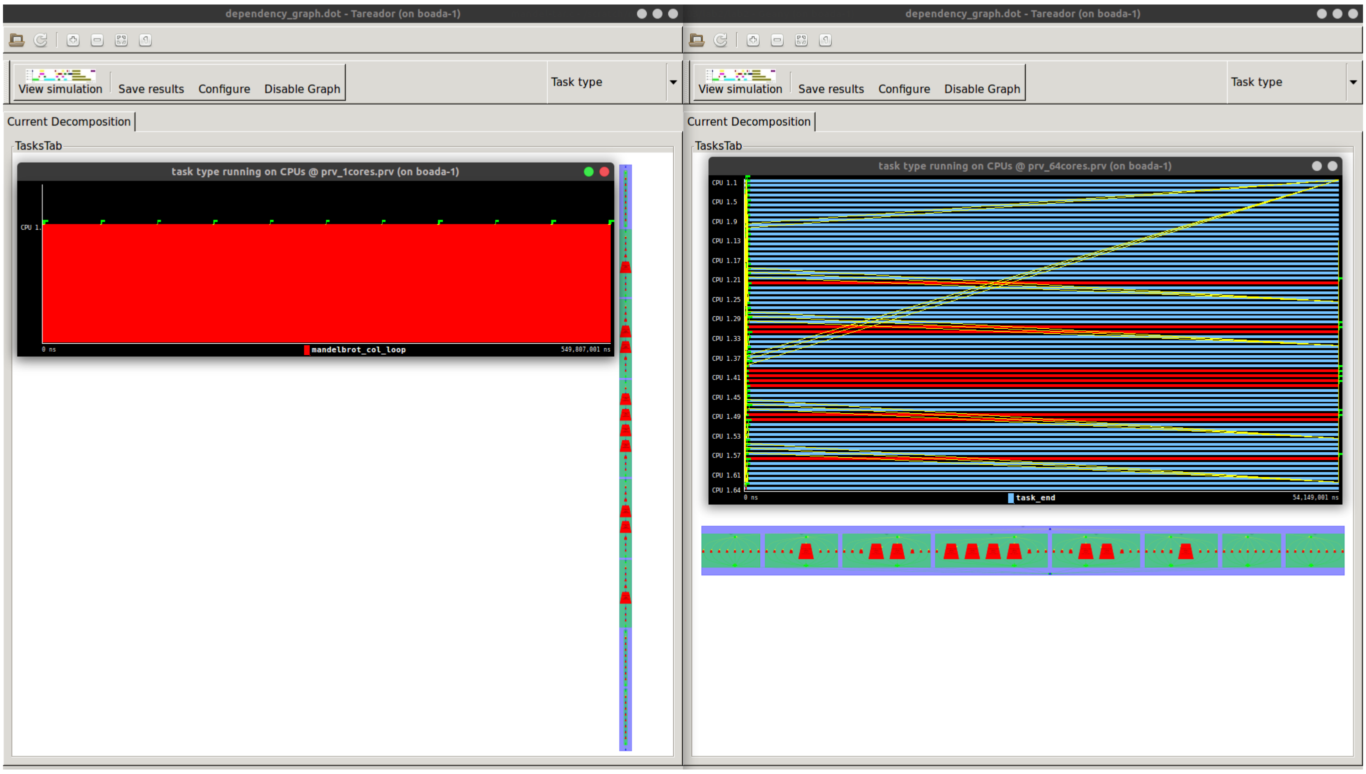
Introduction

In lab 2 we are going to work with Mandelbrot Set. It is a set of complex numbers. With an algorithm to compute Mandelbrot Set we are going to obvserve how to paralelize in diferent ways. Point method and Row method are the methods we are going to work with.



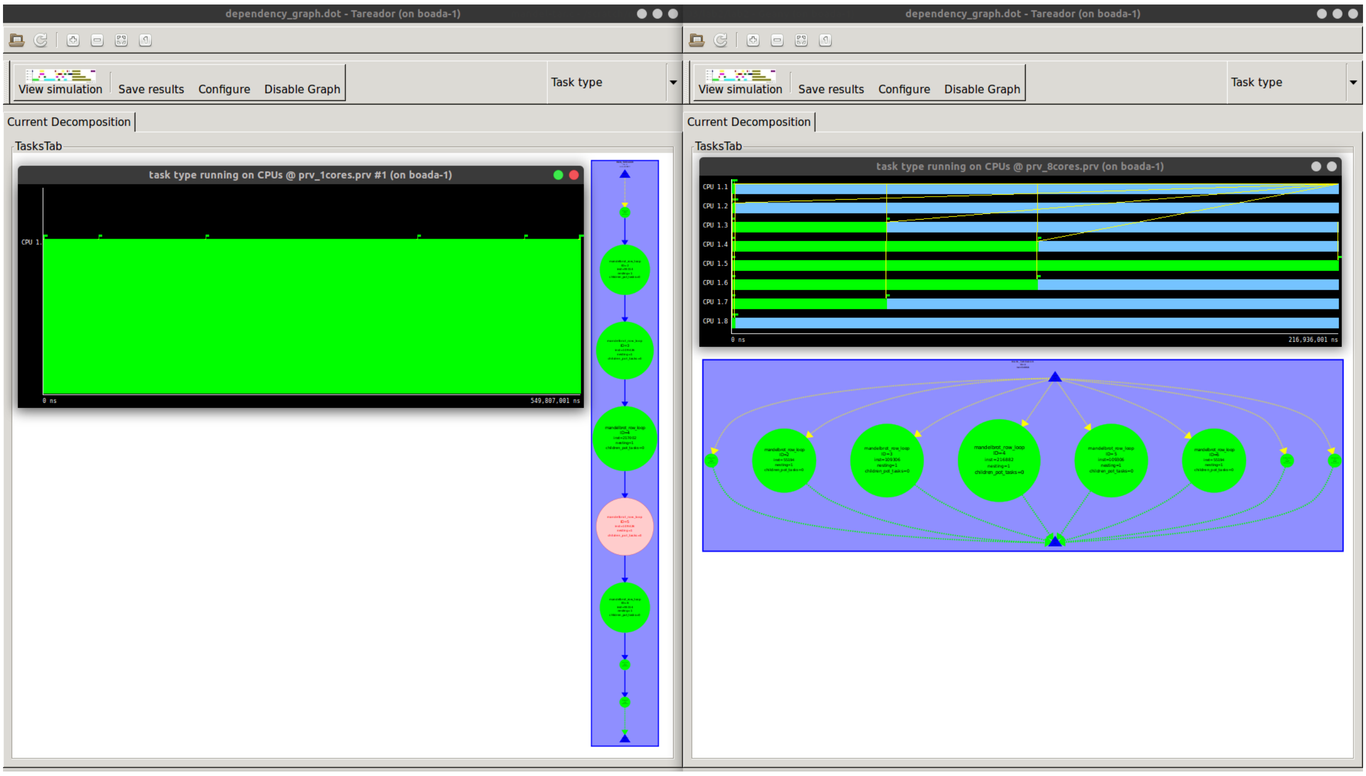
Task decomposition and granularity analysis

First of all, take a look at the task dependence graph of point decomposition. At left is the glaphycal version, we can obvserve that it cannot be paraleliced. However the non display (right window) version is paraleliced. Then, whe can conclude, som part of display code is causing some dependences.



Tareador captures of mandel-tar using point decomposition

Using row decomposition, the result is similar then the previous version. Display version still have the same problem, a data dependence, and non-display version is paralel. Obviously the granularity is bigger. Unlike point decomposition, now we are compting a big number of points with only one task. It is going to reduce overhead time but increasing the granularity.



Tareador captures of mandel-tar using row decomposition

code:

```
(...)
    for (row = 0; row < height; ++row) {
        tareador_start_task("mandelbrot row loop"); //Row decomposition
        for (col = 0; col < width; ++col) {
            tareador_start_task("mandelbrot col loop"); //Point
decompsition

(...)
```

mandel-omp.c

Scale cxolor and display point causes dependency in the graphycal version. It uses vars.

The granularity in point method is smaller and with 64 threads the execution time is much lower than using row method. If you have enough cpus, the point strategy is more adequate, as it allows for a lower time, however, row method it's worth if you have less cores. You will reduce overhead time.

Code protection:

X11 use a variable, named X11_COLO_fake, with dependences. With openMP you can declare critical regions. That protects your code while parallelize from decoherences.

```
(...)
#if _DISPLAY_
            /* Scale color and display point  */
```

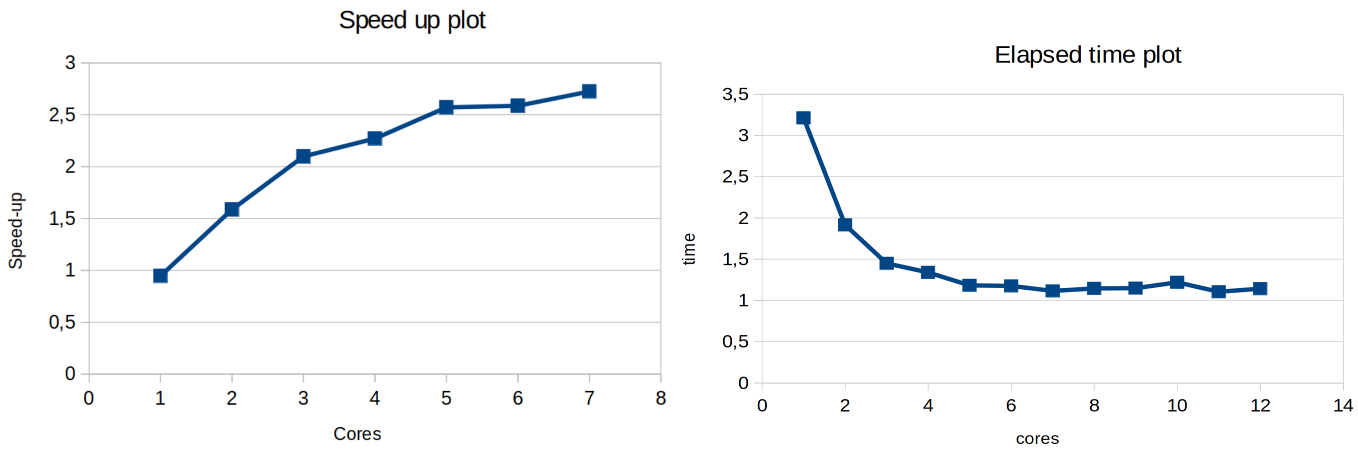
```
            long color = (long) ((k-1) * scale_color) + min_color;
            if (setup_return == EXIT_SUCCESS) {
                                #pragma omp critical (X11)
                                {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
                                }
            }
#else
(...)
```

mandel-omp.c

Point decomposition in OpenMP

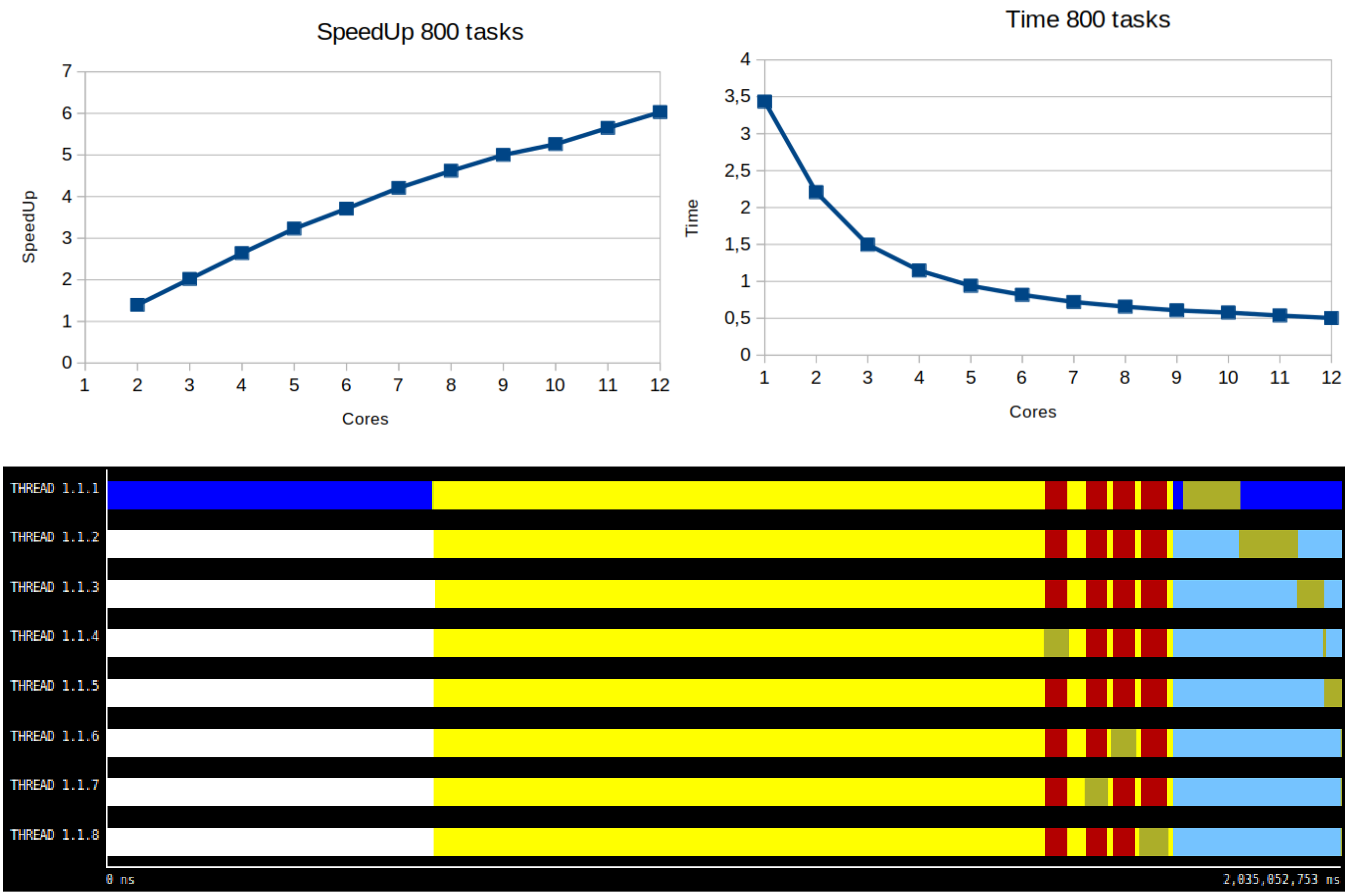
With point decomposition, one task is created for every leaf of the three task sequentilly.

The following plots are the time plot and speed up plot which shows the dependence between time or speed up and the number of cores used while the execution of the program.

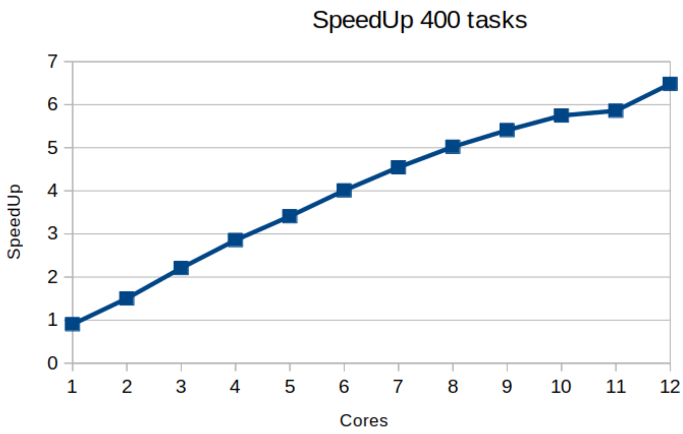


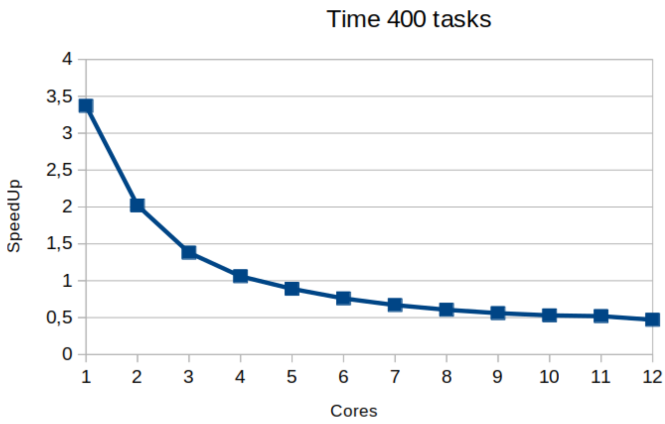
Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program.

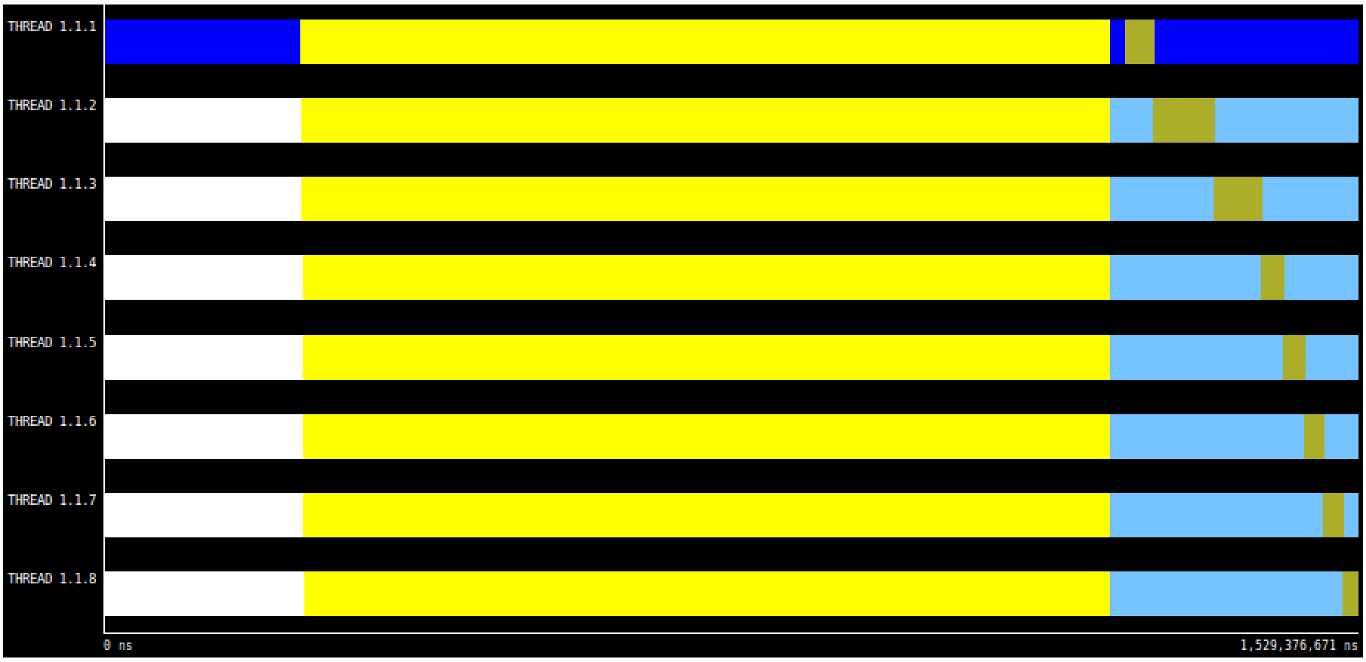
The following plots are representing the relation between different number of tasks created.



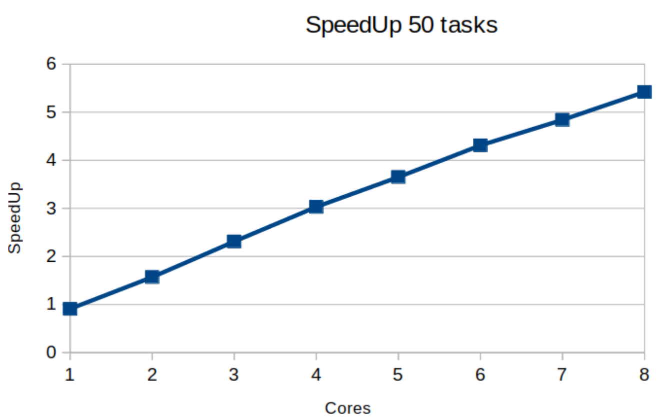
Elapsed time, Speed Up plot and paraver trace for point decomposition version of mandel-omp.c program with 800 tasks.

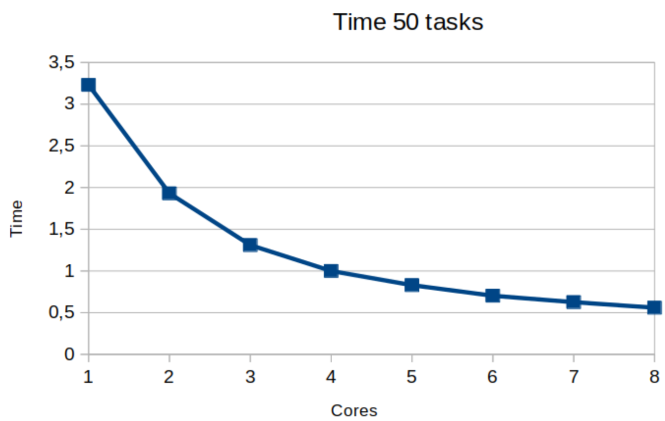


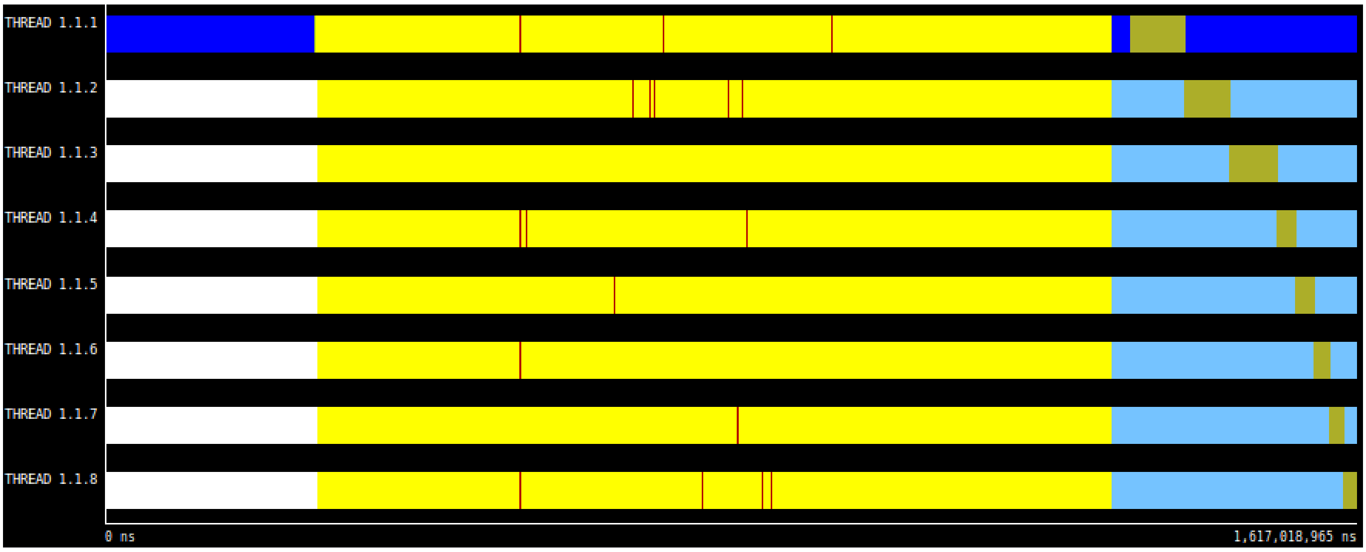




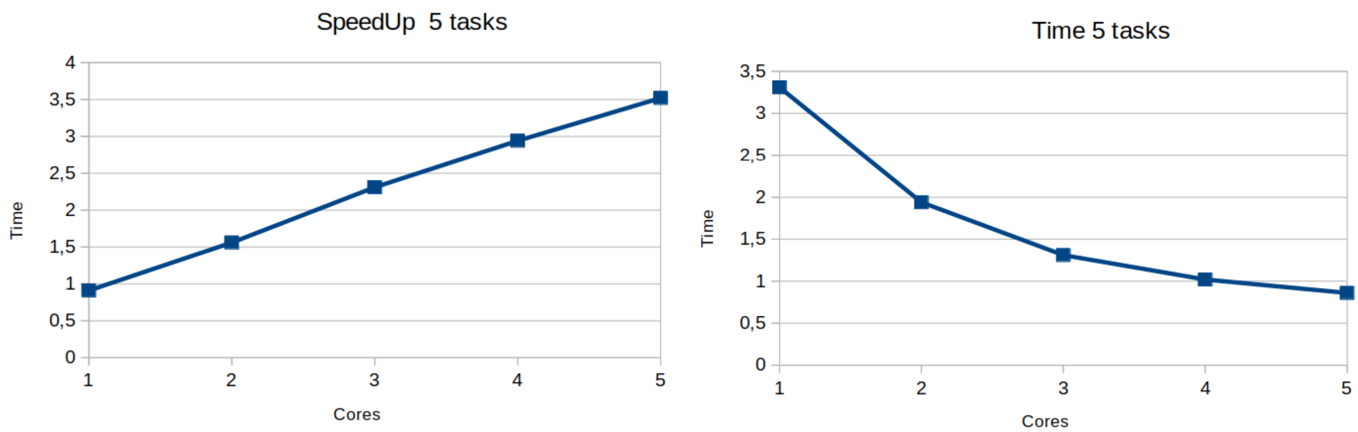
Elapsed time, Speed Up plot for point decomposition version of mandel-omp.c program with 400 tasks.

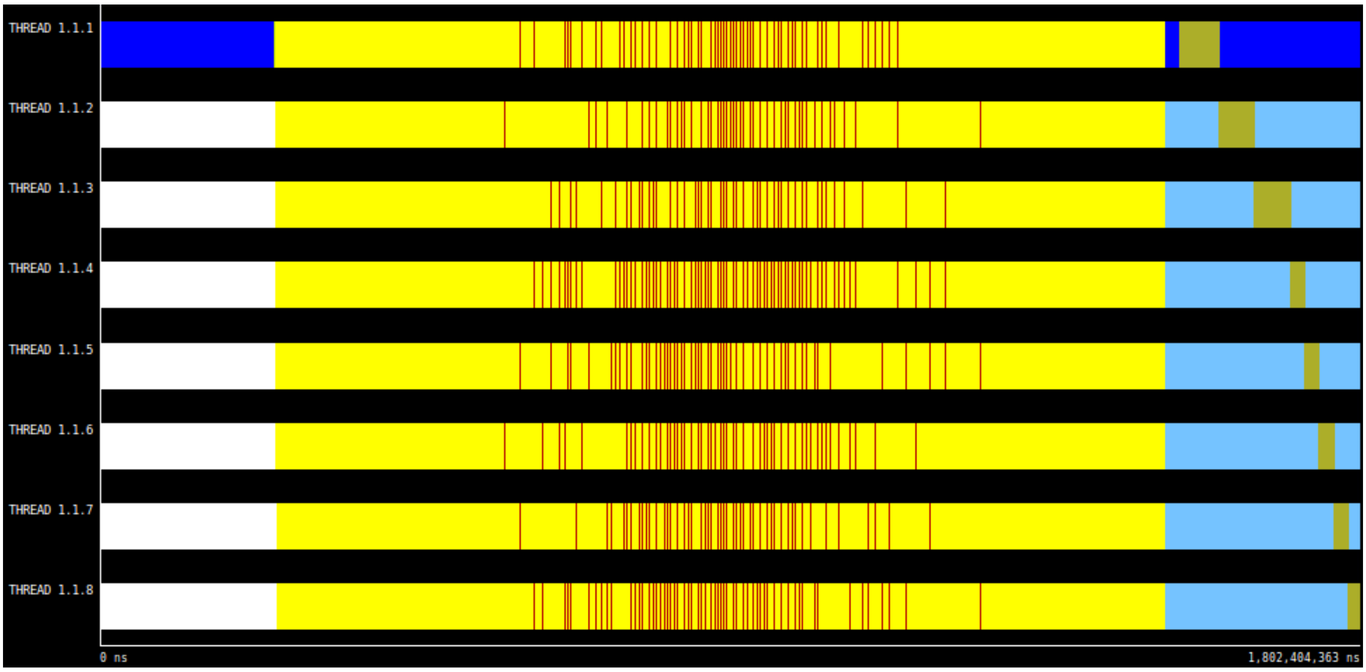




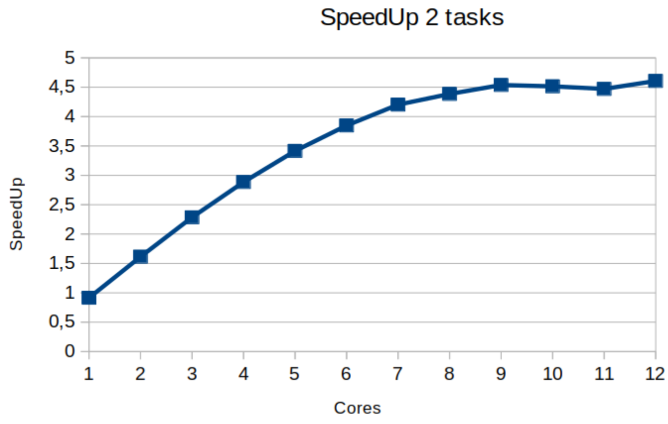


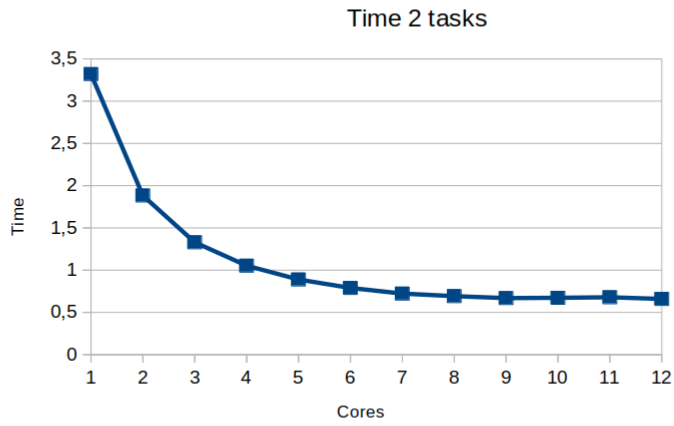
Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 50 tasks.

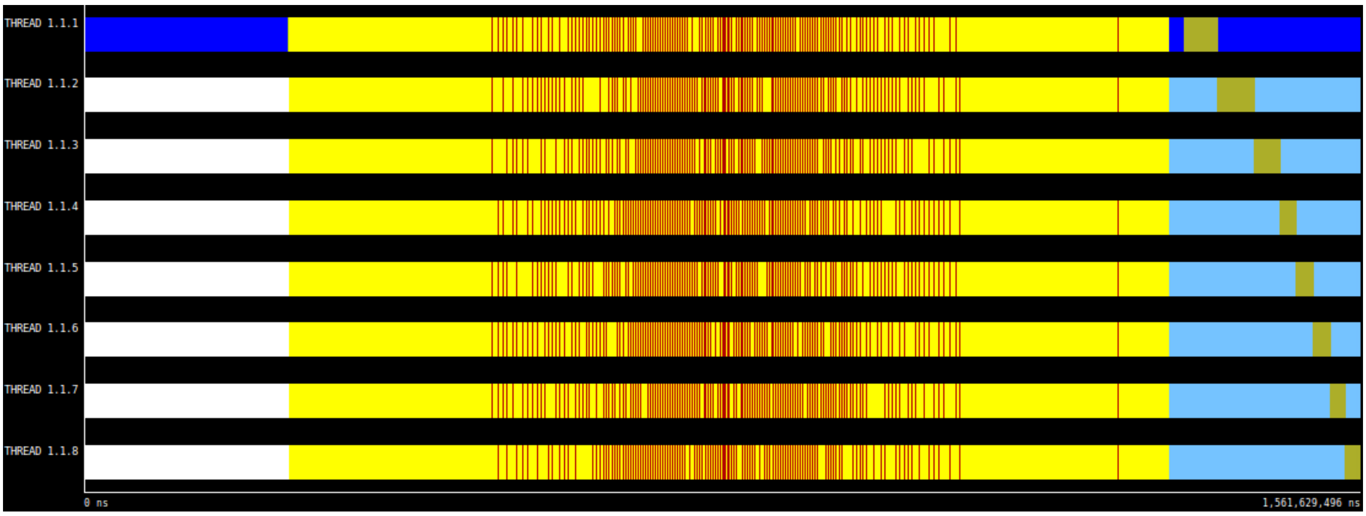




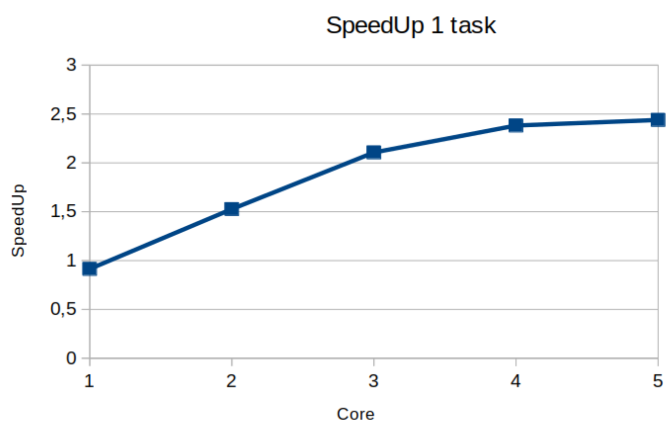
Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 5 tasks.

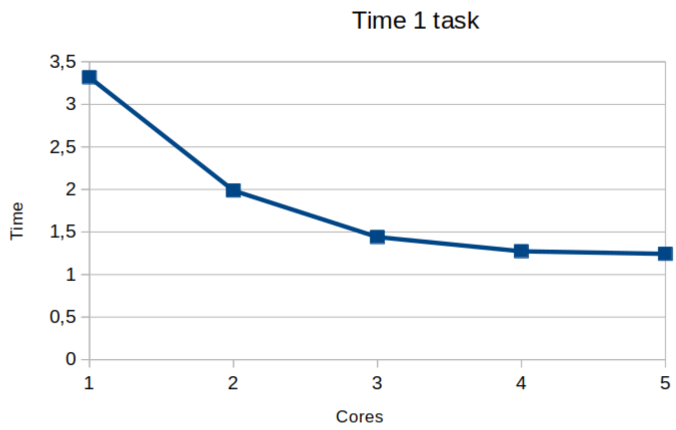


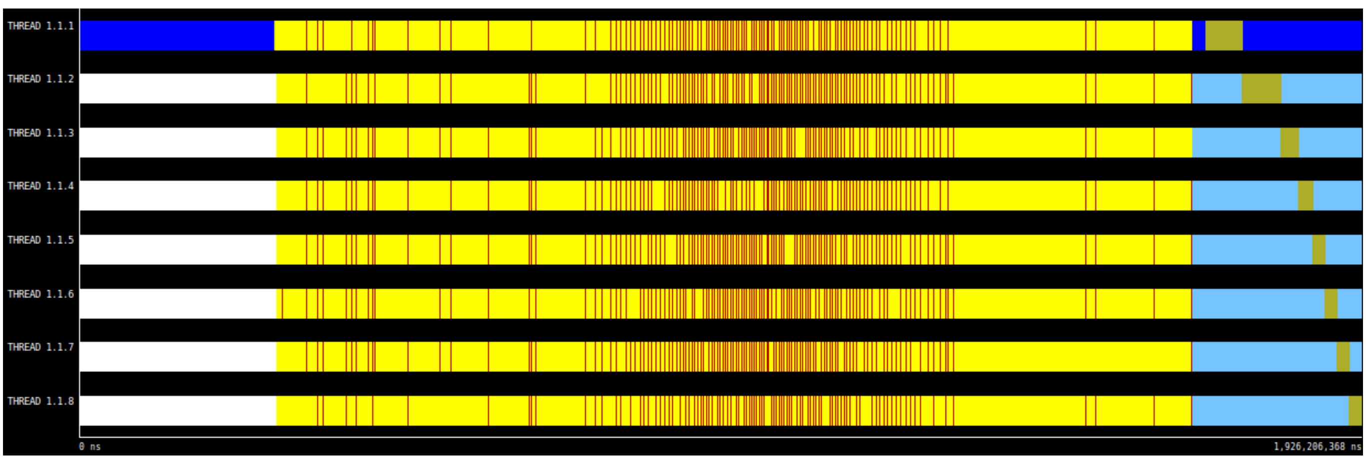




Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 2 tasks.







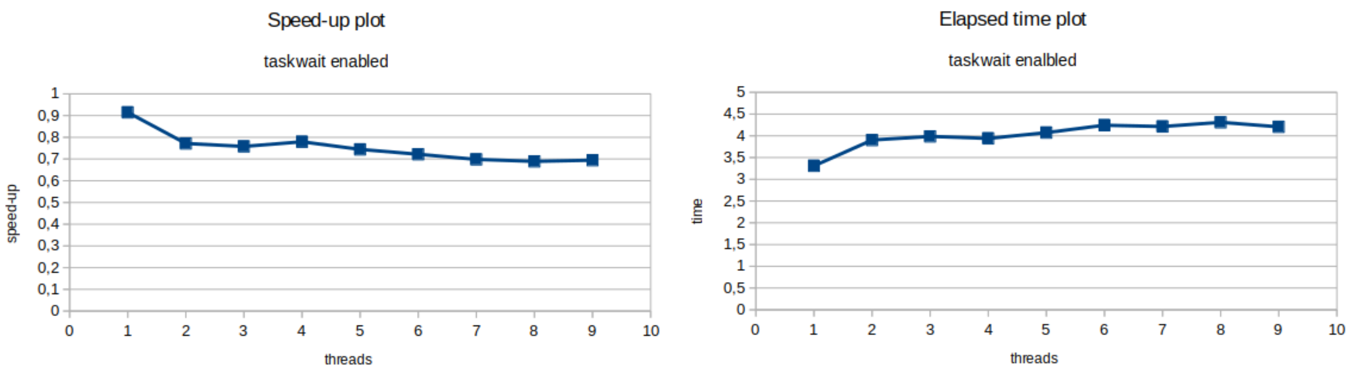
Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 1 tasks.

It appears that after 8 threads the speed-up and the execution time plots begin to normalize. Thats cause the mandelbrot only saves a great portion of execution time until 8 cores, after that, we only get small bonuses.

Speed up doesn’t increase any more, then time also doesn’t increase. We can conclude the maximum number of cores that the program can deal with is 5 in that kind of palatalization.

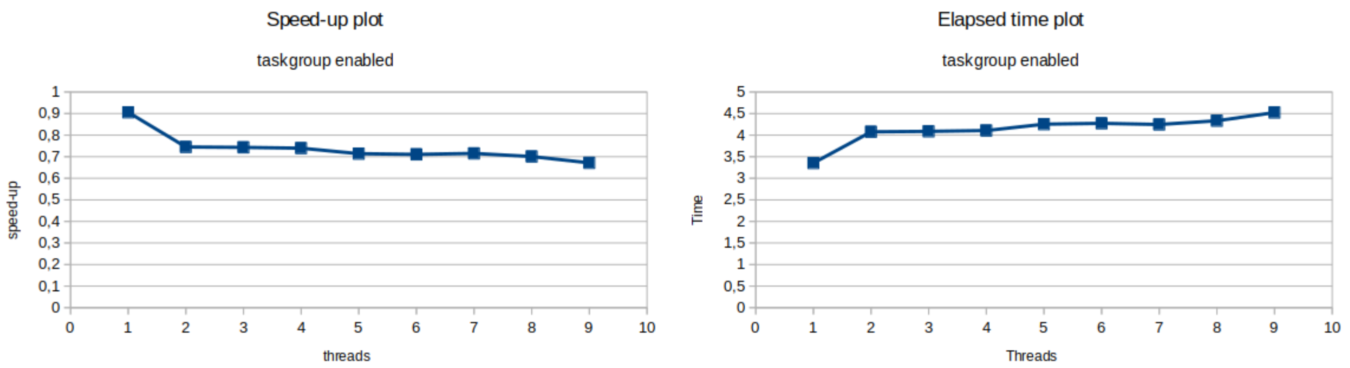
Optional: taskgroup vs. taskwait

With taskwait enabled, we expirience a decrese in performance, thats due to the overheads of creating every task, and having to wait for each of them to finish. The more threads, the more tasks are in paralell, therefore the greater is the waiting time. It can be appreciated below.



Speed-up and elapsed time plot of mandel-omp using taskwait

For the same reason of taskwait, taskgroup also introduces a deacrising performance. There is almost no difference between the two becouse in this program, the loops we are putting on pararell don’t create other tasks in them. This way, using both of the methods (taskwait and taskgroup) gives the same result.



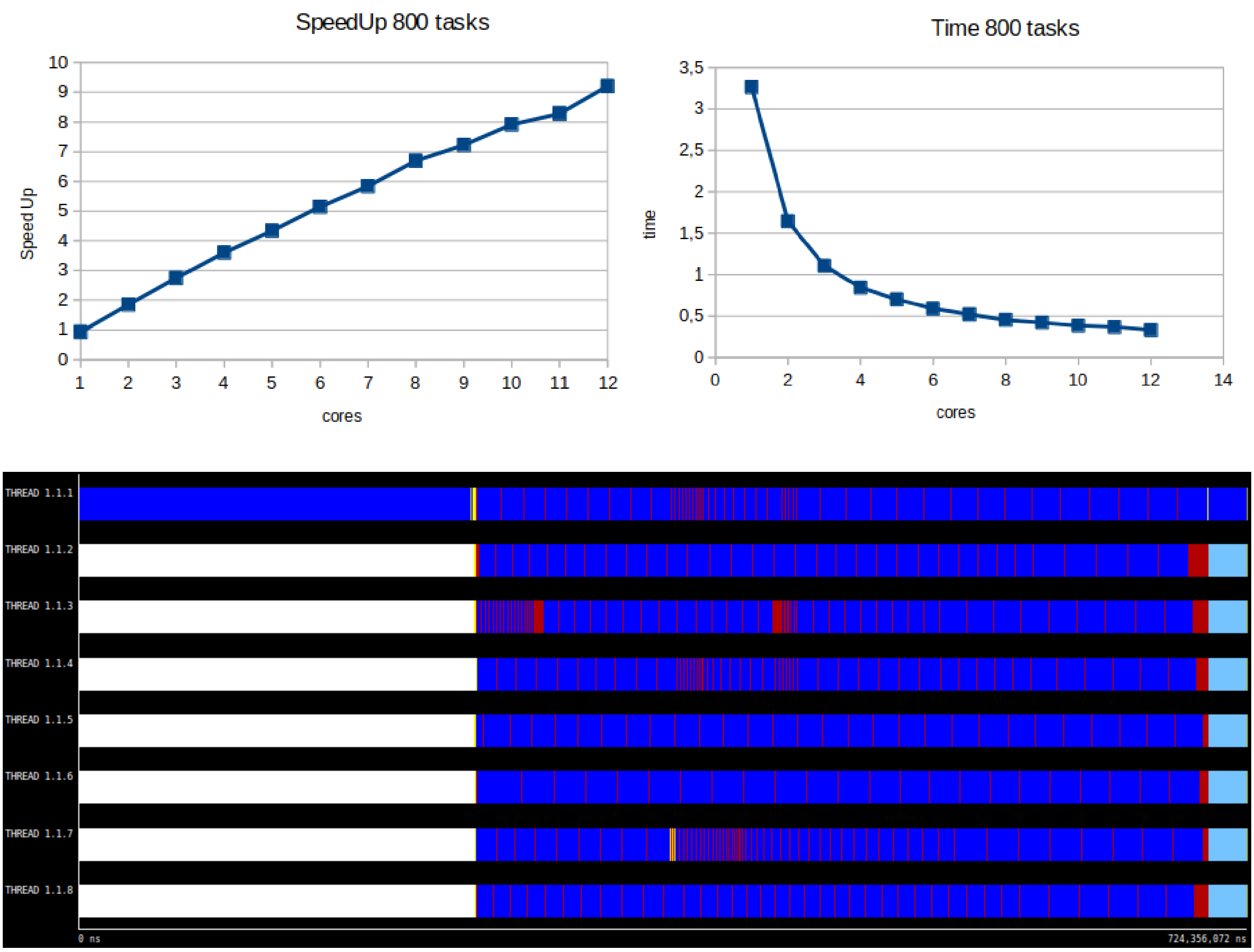
Speed-up and elapsed time plot of mandel-omp using taskgroup

Row decomposition in OpenMP

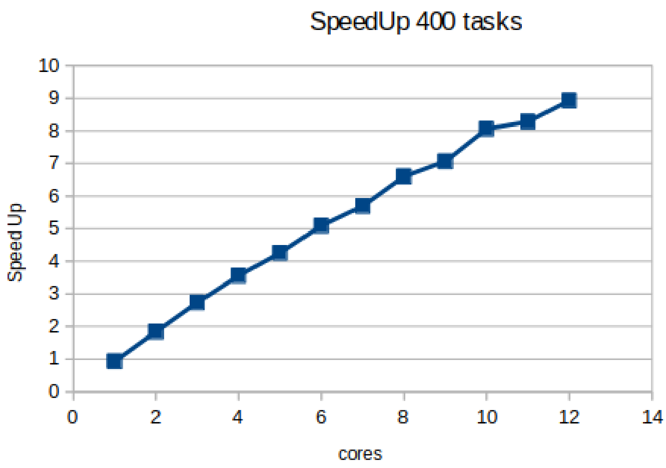
We have used parallelization pragmas in the row loop. Due to that we got a task descompostion based in rows. This helps reducing overhead time added to the executable. However that could translate in a greater execution time. We would see this with the analysis of scalability.

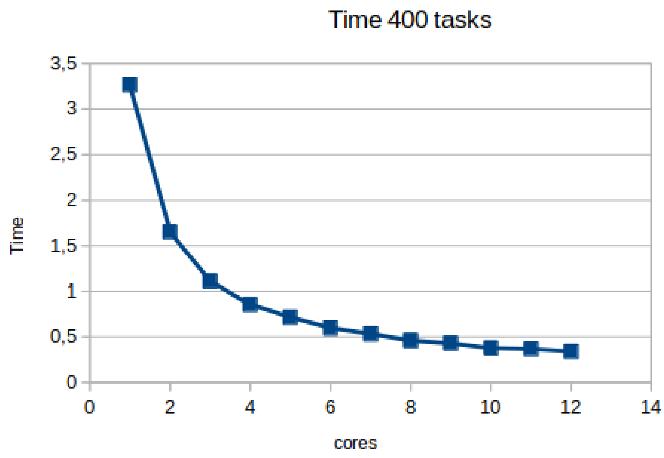
```
(...)
    #pragma omp parallel
        #pragma omp single
        #pragma omp taskloop firstprivate(row) num_tasks(800)
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            complex z, c;
(...)
```

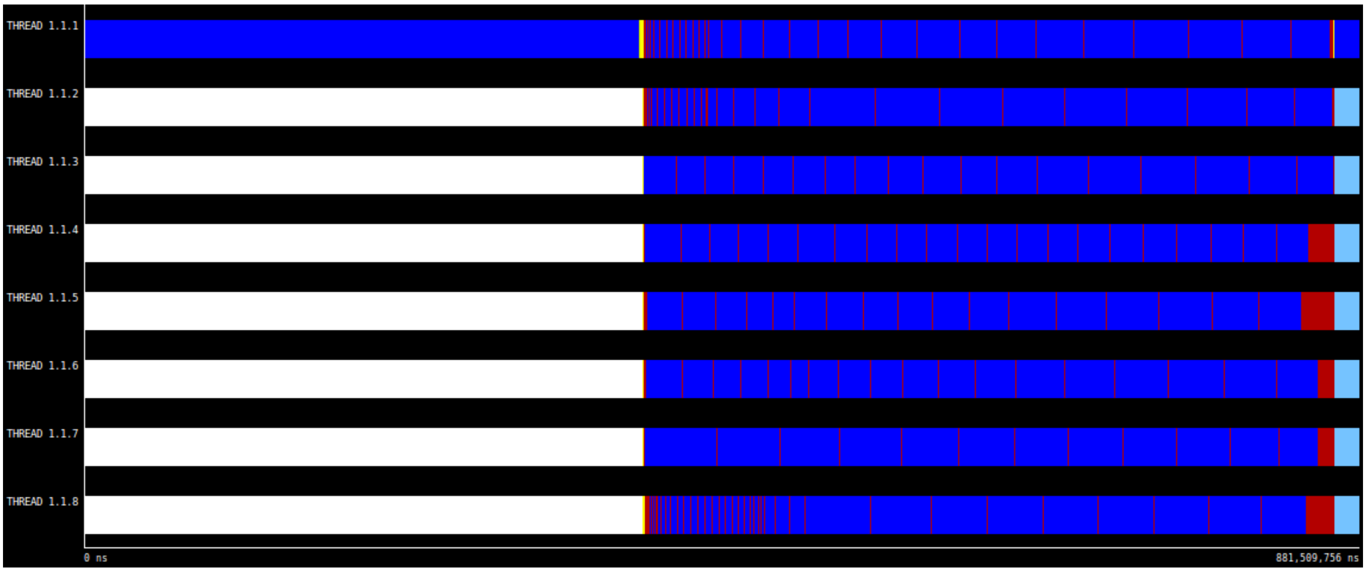
mandel-omp.c



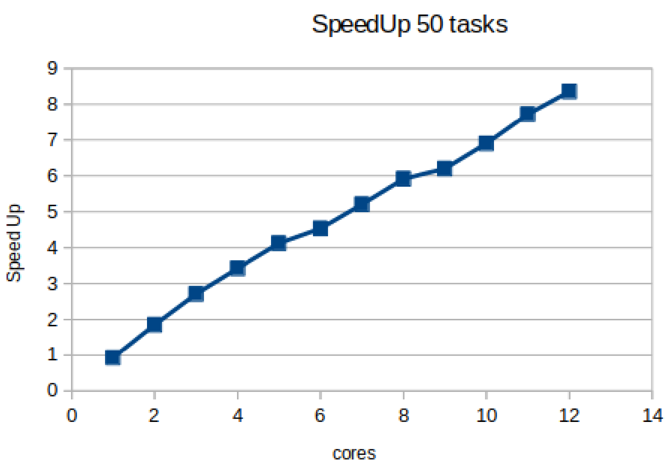
Elapsed time, Speed Up plot and paraver trace for point decomposition version of mandel-omp.c program with 800 tasks.

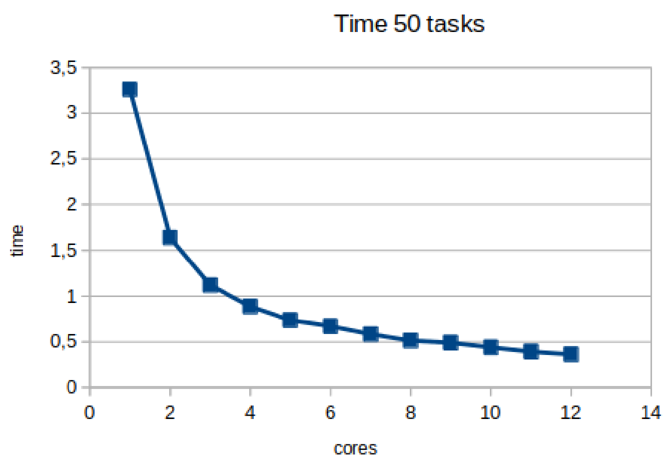


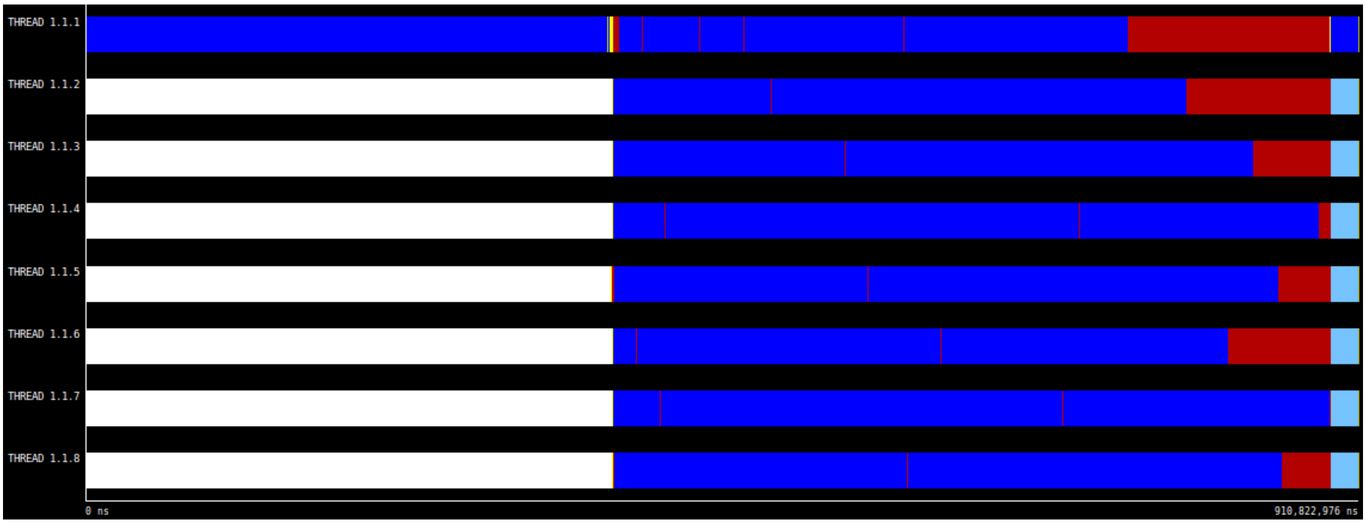




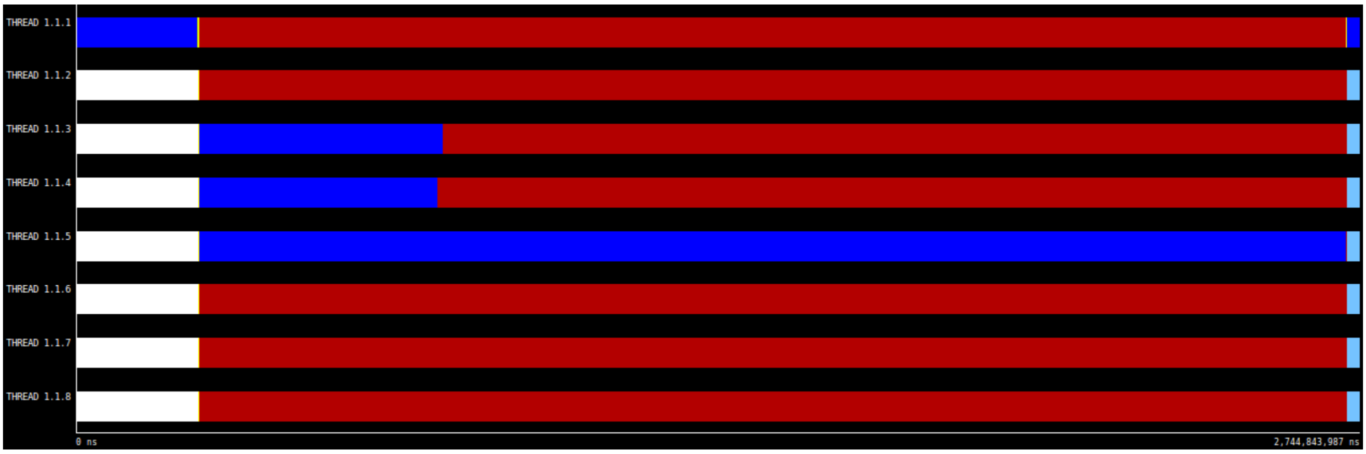
Elapsed time, Speed Up plot for point decomposition version of mandel-omp.c program with 400 tasks.



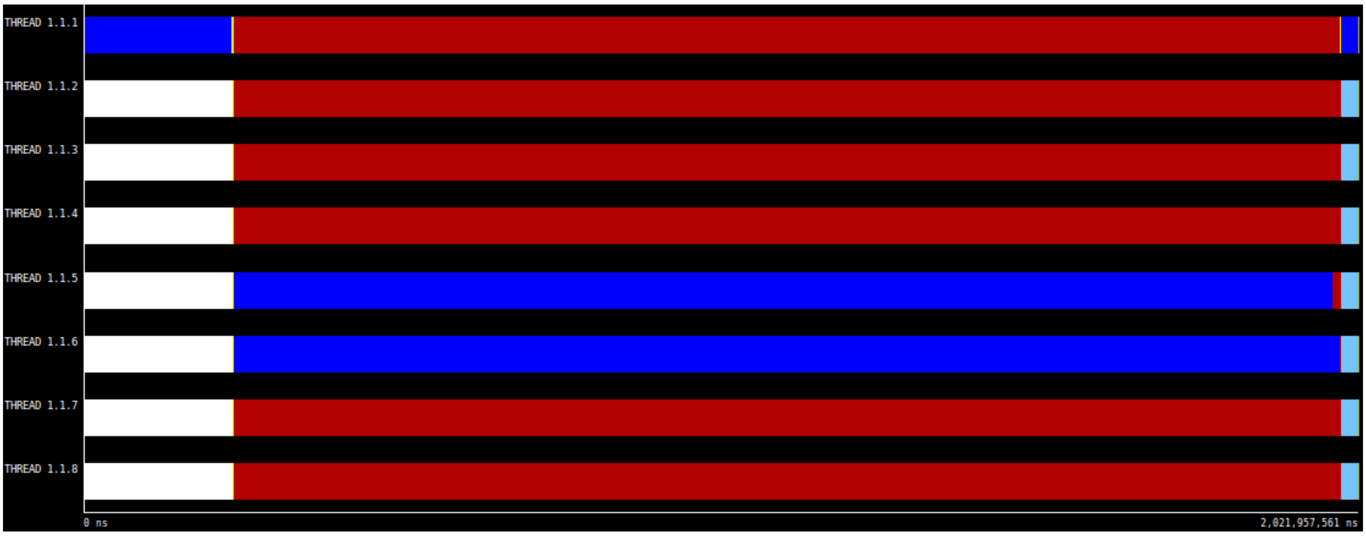




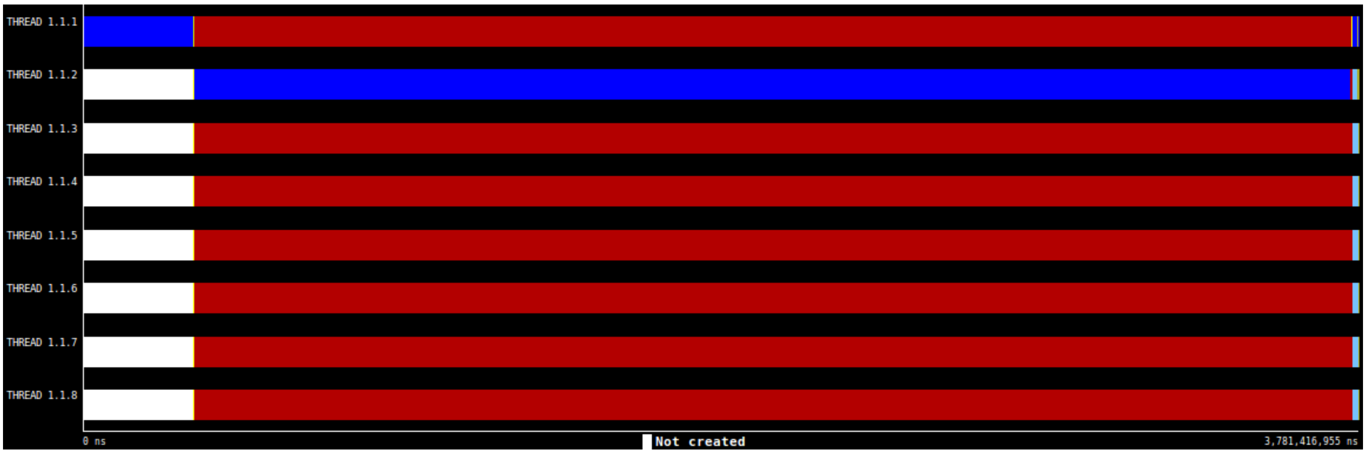
Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 50 tasks.



Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 5 tasks.



Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 2 tasks.



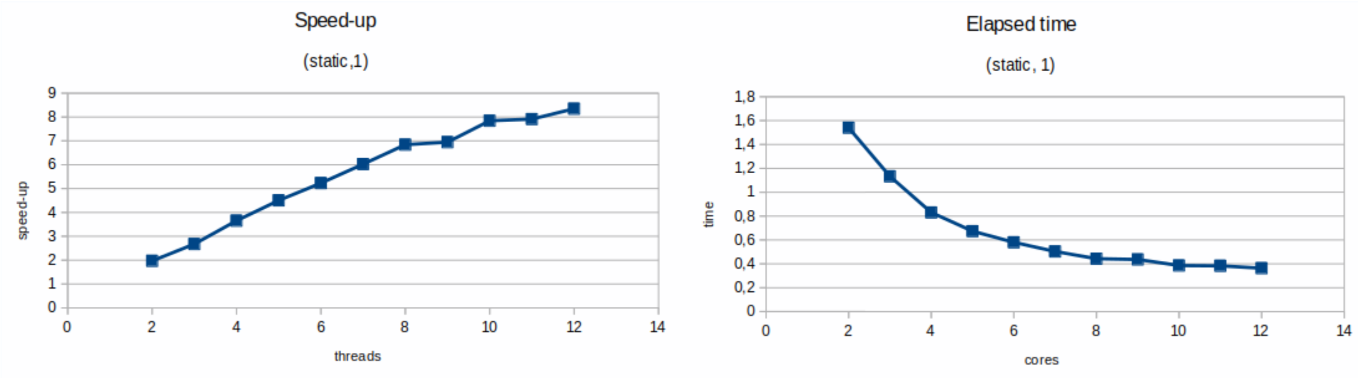
Elapsed time and Speed Up plot for point decomposition version of mandel-omp.c program with 1 tasks.

Optional: for–based parallelization

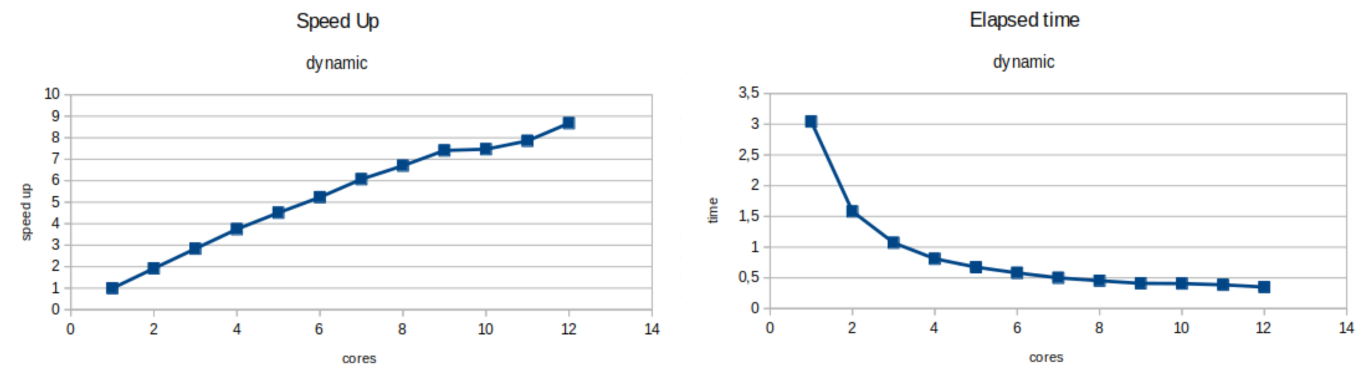
Code:

```
(...)
for (int row = 0; row < height; ++row) {
    #pragma omp parallel
    #pragma omp for
    for (int col = 0; col < width; ++col) {
(...)
```

mandel-omp.c Now, we are going to observe speed up and elapse time plots of mandel algorithm using differents kinds of directives.

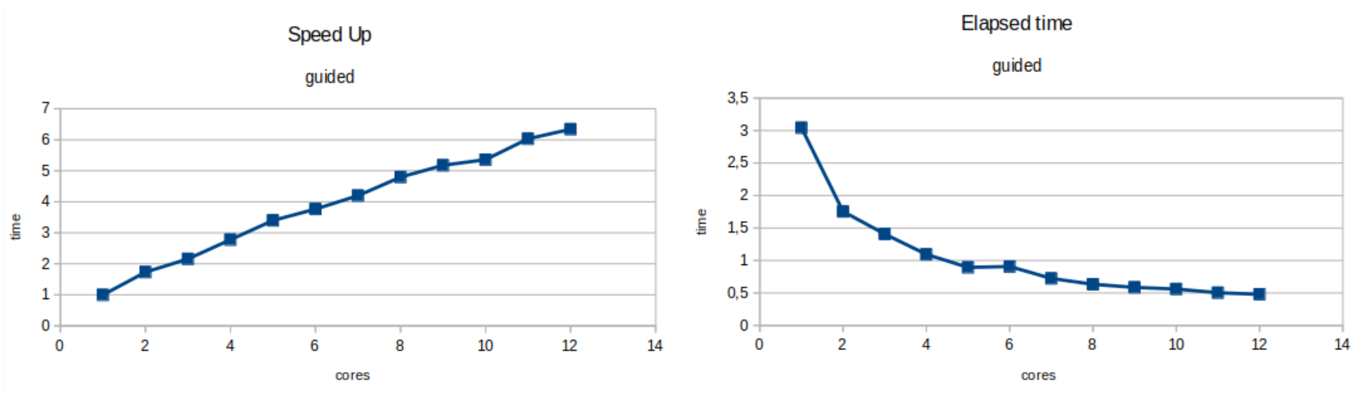


Speed up and elapsed time plots of mandel with omp for static directive



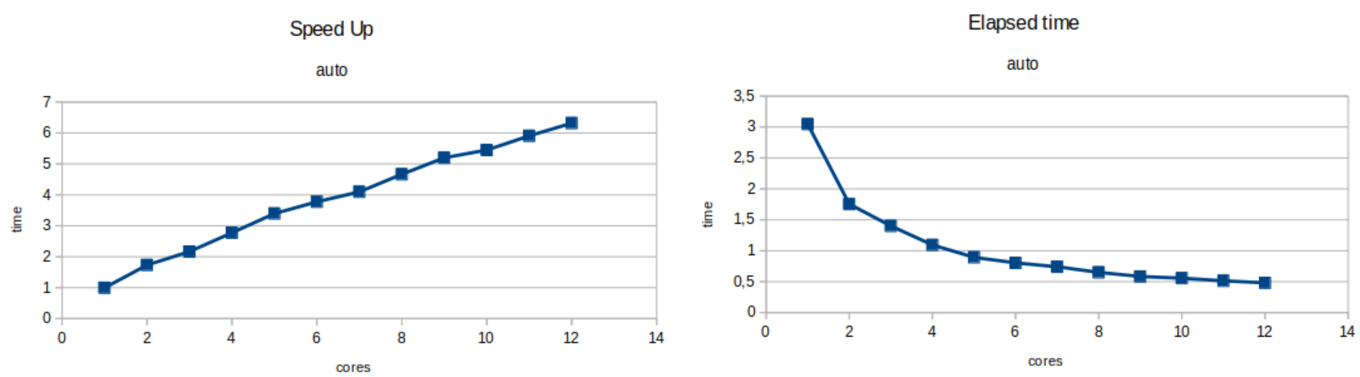
Speed up and elapsed time plots of mandel with omp for dynamic directive

The nearly identical result of the performance of dynamic and (static,1) may be cause in this program the dynamic schedule choses 1 chunck.



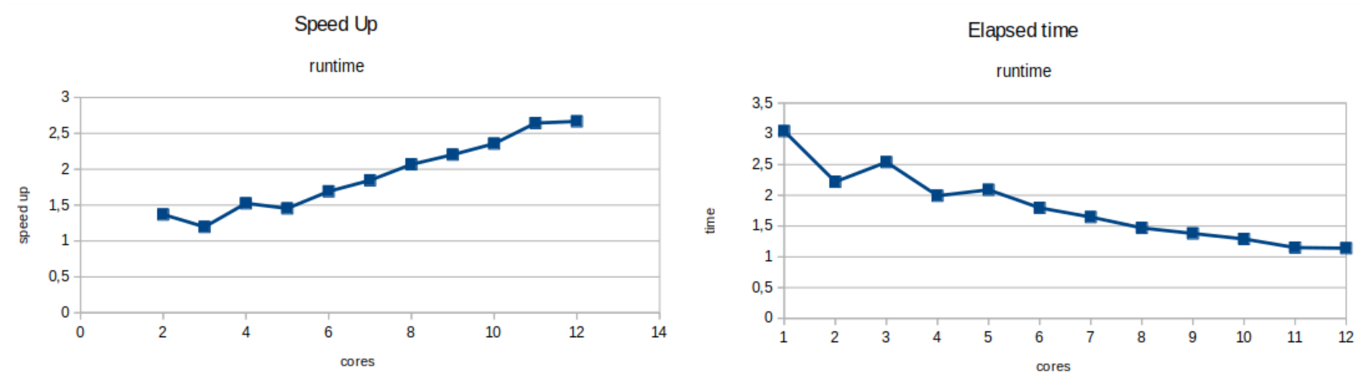
Speed up and elapsed time plots of mandel with omp for guided directive

Guided option performs worst than the static and dynamic. It could be because it create bigger chunks.



Speed up and elapsed time plots of mandel with omp for auto directive

That one have very similar performance than the previous one, we can suppose that auto selects guided directive.



Speed up and elapsed time plots of mandel with omp for runtime directive

The last one performs even worst than guided. That's because the chunk size is calculated during the execution.