Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

*2018-2019 Q2*

**par2110**

Maurici Abad Gutierrez

Nicolas Schrack

# Introduction

The Mandelbrot set is the set of complex numbers c for which the function fc(z) = z2+c does not diverge when iterated from z = 0, i.e., for which the sequence fc(0), fc(fc(0)), etc., remains bounded in absolute value. In this report we are going to analyze a code that implemente that set with some implementations and tools.

To analyze the code and its paralization we will use Tareador which will help us to see the dependences of the parameters and other relevant information like the number of executions will each task do. Another procedure that will help us is an instrumented analisy, which will be implemented with the help of some Scripts.

To understand the instrumented analyze we will use Paraver, a helpful tool that will show us some important information in a graphicaly way.

# Task decomposition and granularity analysis

## Codes

The Tasks are defined at granularity level of Point:

/\* Calculate points and save/display \*/

for (row = 0; row < height; ++row) {

for (col = 0; col < width; ++col) {

**tareador\_start\_task("point");**

// calculation of a single point

**tareador\_end\_task("point");**

}

}

The tasks are defined at granularity level of Row:

/\* Calculate points and save/display \*/

for (row = 0; row < height; ++row) {

**tareador\_start\_task("row");**

for (col = 0; col < width; ++col) {

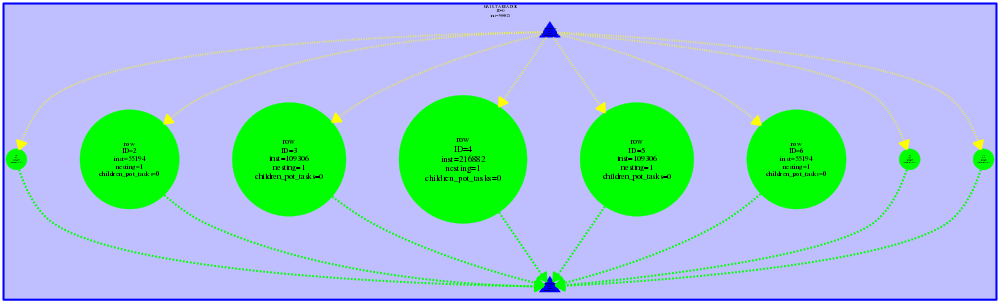
// calculation of a single point

}

**tareador\_end\_task("row");**

}

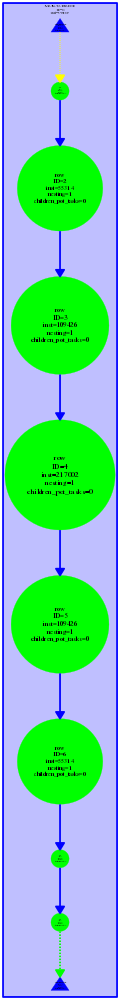
## Dependency Graphs



Row Normal (*mandel-tar)*



Point Normal (*mandel-tar)*



Row Display (*mandeld-tar)*



Point Display (*mandeld-tar)*

## Analysis

The non-graphical version *mandel-tar* has two common characteristics:

1. Both are embarrassingly paralelizable, because there are no dependencies.
2. Tasks have notable different sizes, because some points diverge faster than others.

The graphs are so different because the *mandeld-tar* is painting the pixel just after the point is calculated, and the library that paints the pixels uses a variable that is shared, so it can’t run in parallel due to data erase.

The problem is here:

#if \_DISPLAY\_

/\* Scale color and display point \*/

long color = (long) ((k-1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

**XSetForeground (display, gc, color);**

**XDrawPoint (display, win, gc, col, row);**

}

#else

output[row][col]=k;

#endif

The solution that requires less changes is using #pragma omp critical in the section that paints each pixel. We also thought on splitting the word: calculate first the points and when you have all of them paint them. Although we think that this second option is more efficient, it would take a lot more changes to the code, so we won’t do it.

## Choosing the right granularity

Row : 25.578360s with max. iterations 1000

Point: 48.821805s with max. iterations 1000

The granularity of the Row is better than the Point because of the overhead.

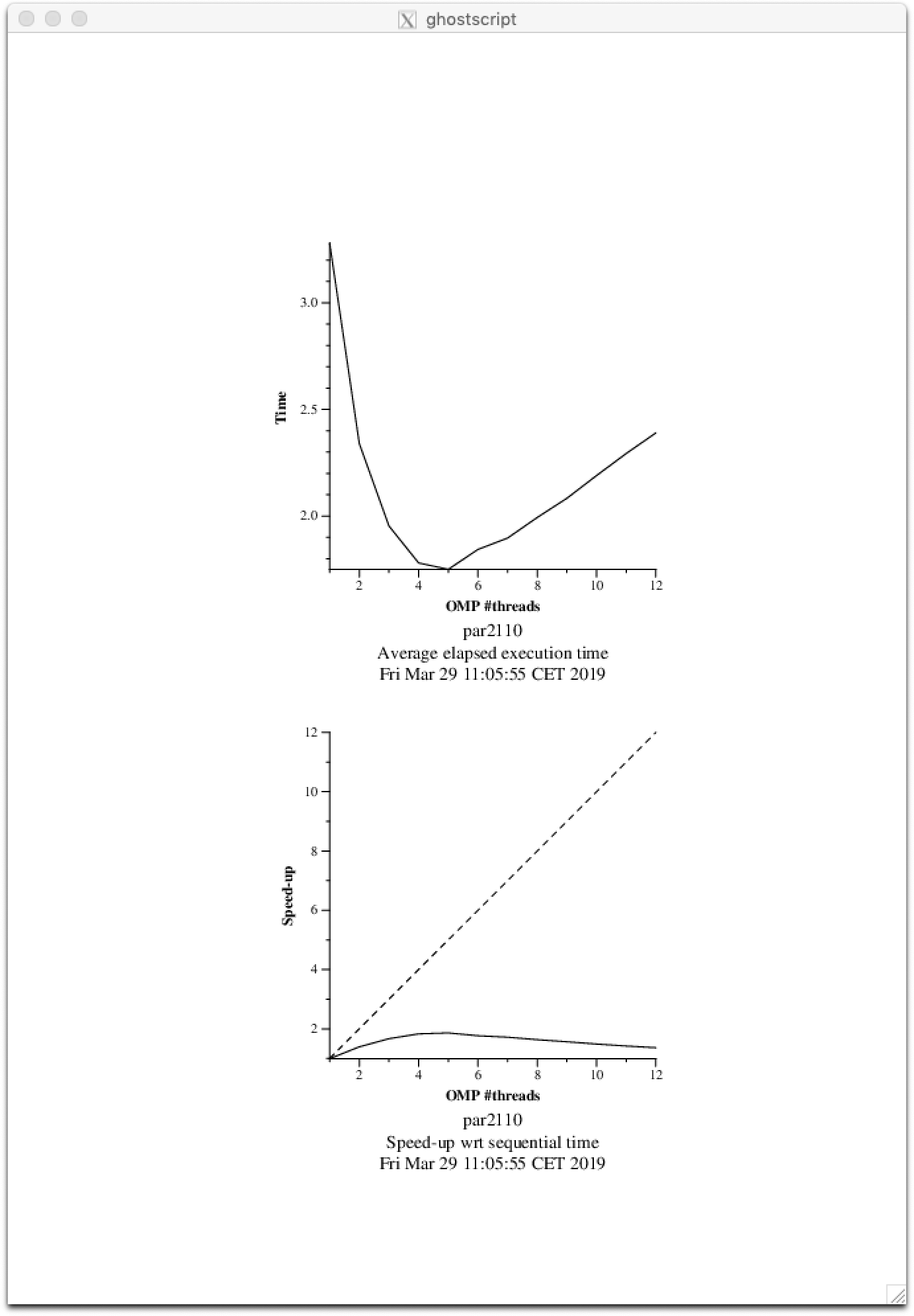
# 

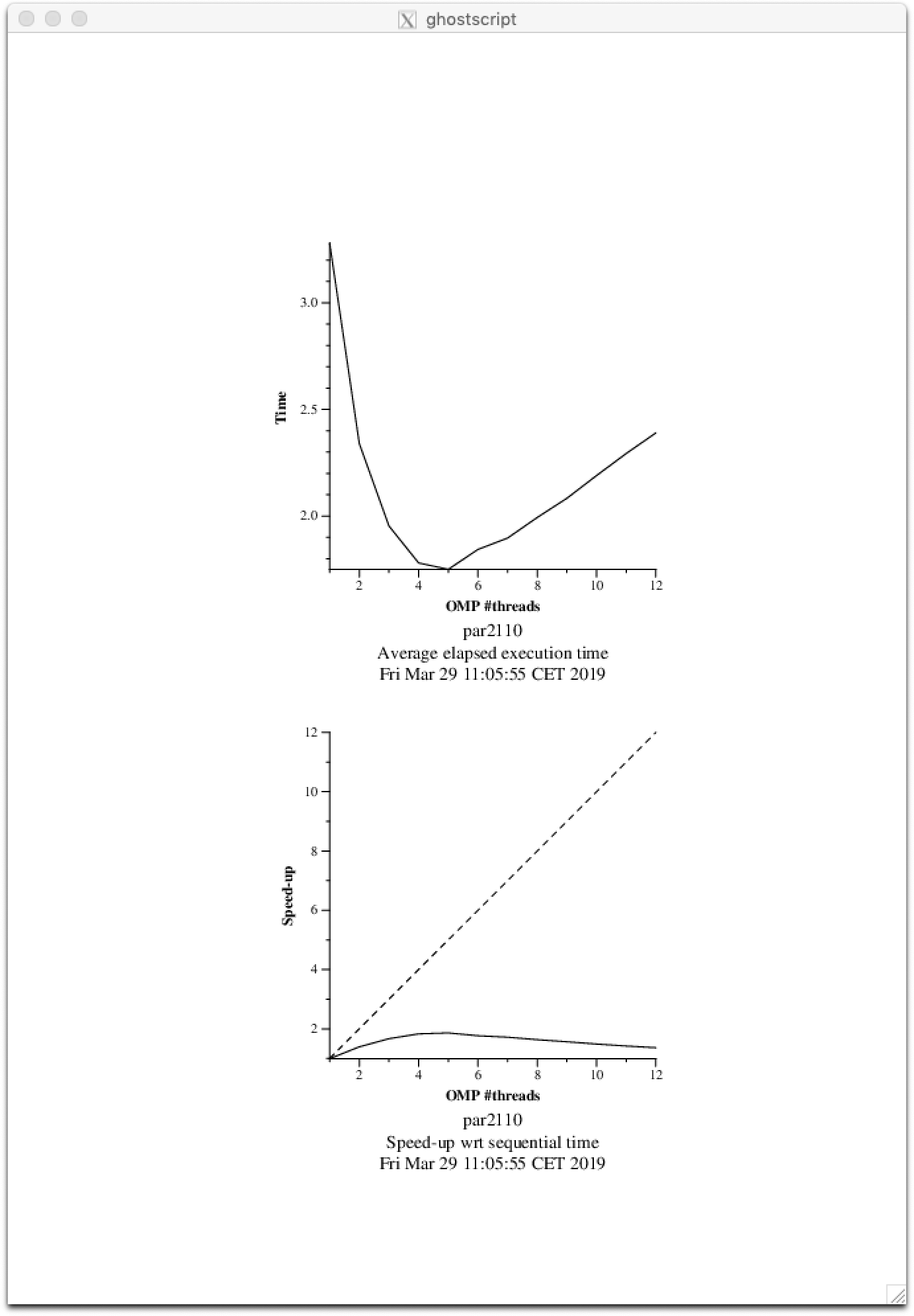
# Point decomposition in OpenMP

## 1. Version (Task)

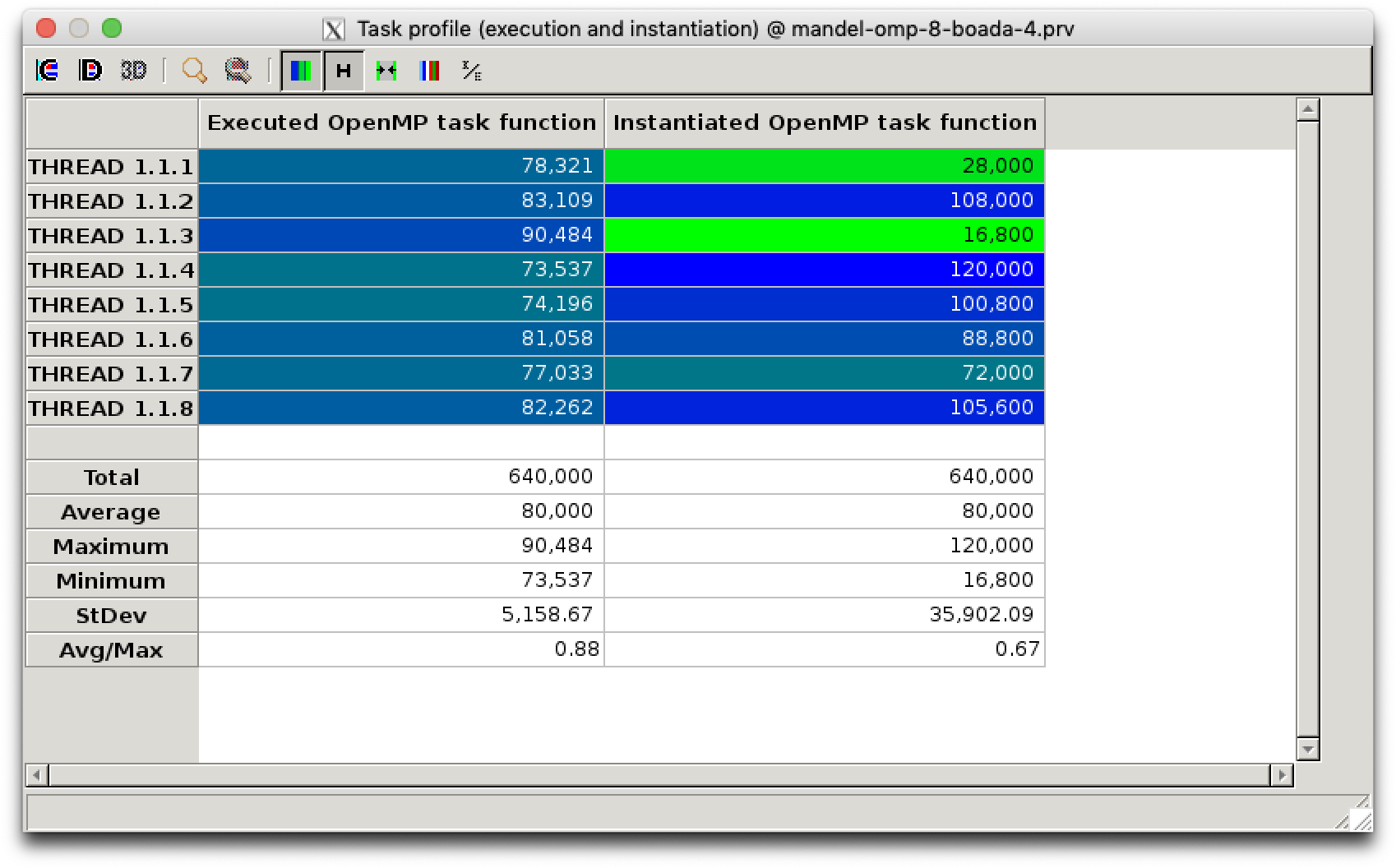
|  |
| --- |
| for (row = 0; row < height; ++row) {  #pragma omp parallel  #pragma omp single  for (col = 0; col < width; ++col) {   #pragma omp task firstprivate(col)  {  ... |

1. The image is correct and the sequential version is faster. The parallel execution with one thread took 3.326357s and the sequential execution took 3.285090s.  
   It takes more time due to overheads, that the sequential doesn’t have to do.  
   With 8 processors the images is not correct, it exits with a segmentation fault error.
2. Both outputs are the same. The scalability is wrong because it takes more time as the number of processors increases.



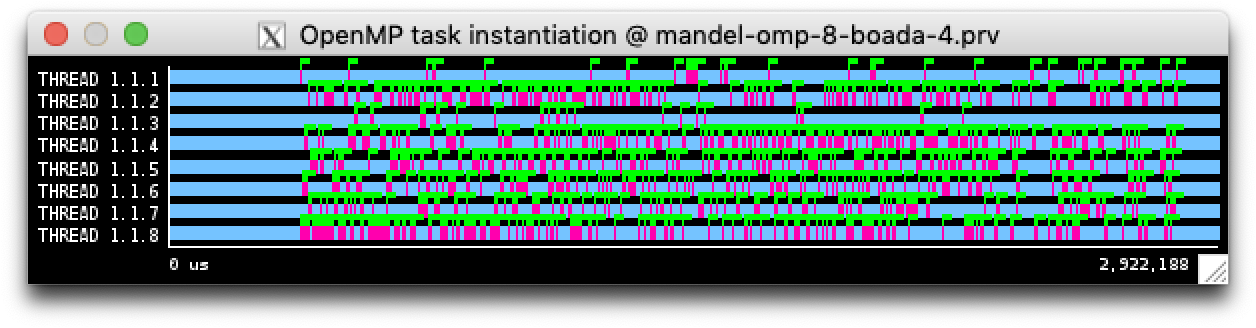


1. 640.000 tasks are being created and executed.   
   Paraver doesn’t show when a specific task is executed, but we can know how many tasks are created and executed by each thread.   
   As we can see in the image, the first and second threads create very few tasks, and the fourth creates a lot more. But all the threads execute approximately the same number of tasks, the variations depend on the number of tasks that the thread created. The parallel construct is invoked 800 times (the value of the height variable). The single worksharing construct is invoked 800 times (the value of the height variable). We looked at the code to figure it out.

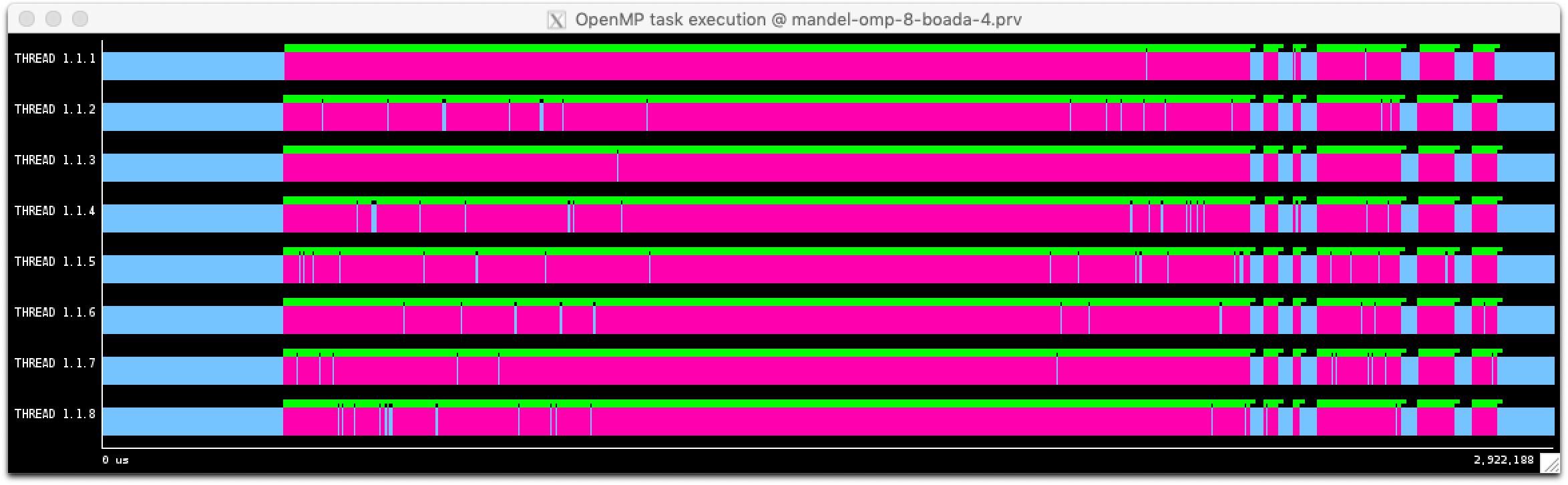




Task execution timeline of mandel-omp-8-boada-4



Task execution timeline of mandel-omp-8-boada-4



Task execution timeline of mandel-omp-8-boada-4



Task execution timeline of mandel-omp-8-boada-4

## 2. Version (Task with Taskwait)

|  |
| --- |
| #pragma omp parallel  #pragma omp single  for (row = 0; row < height; ++row) {  for (col = 0; col < width; ++col) {  #pragma omp task firstprivate(row, col)  {  ...  }  } #pragma omp taskwait |

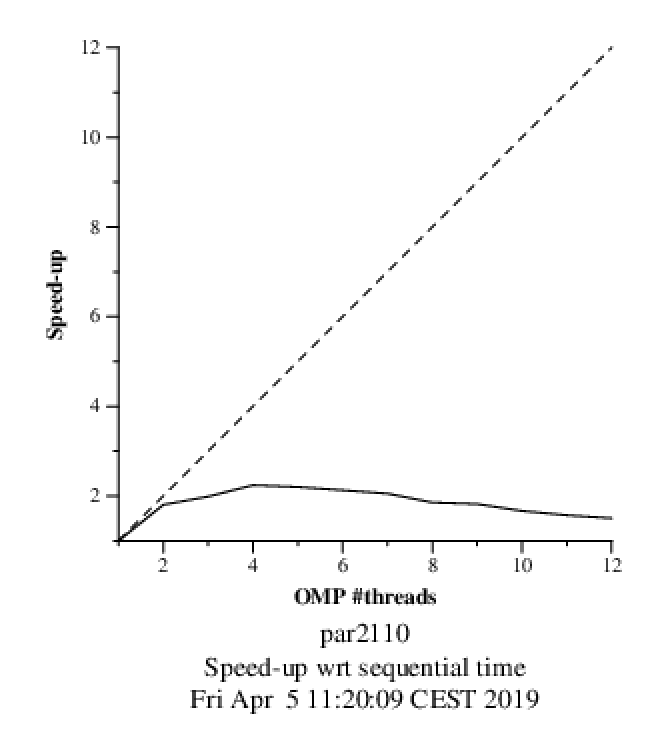
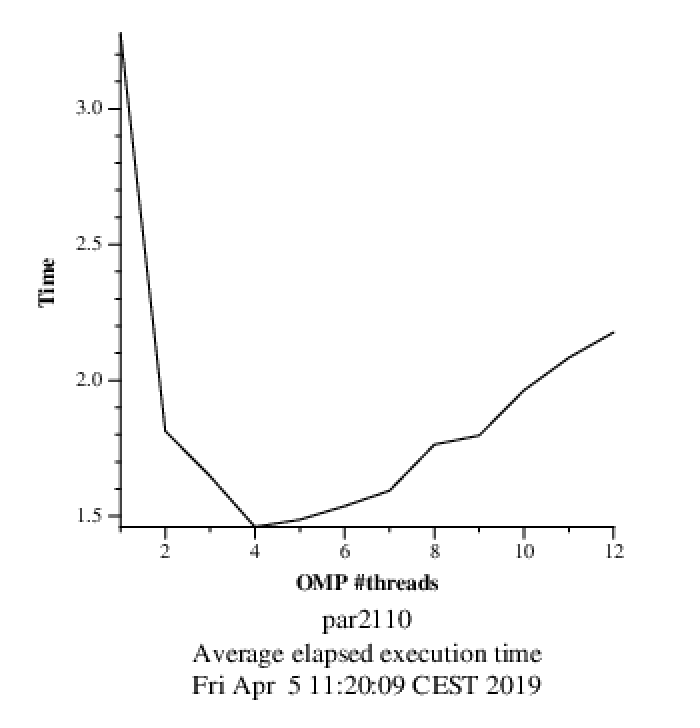
If we change the code it will wait until each point finishes at each row.

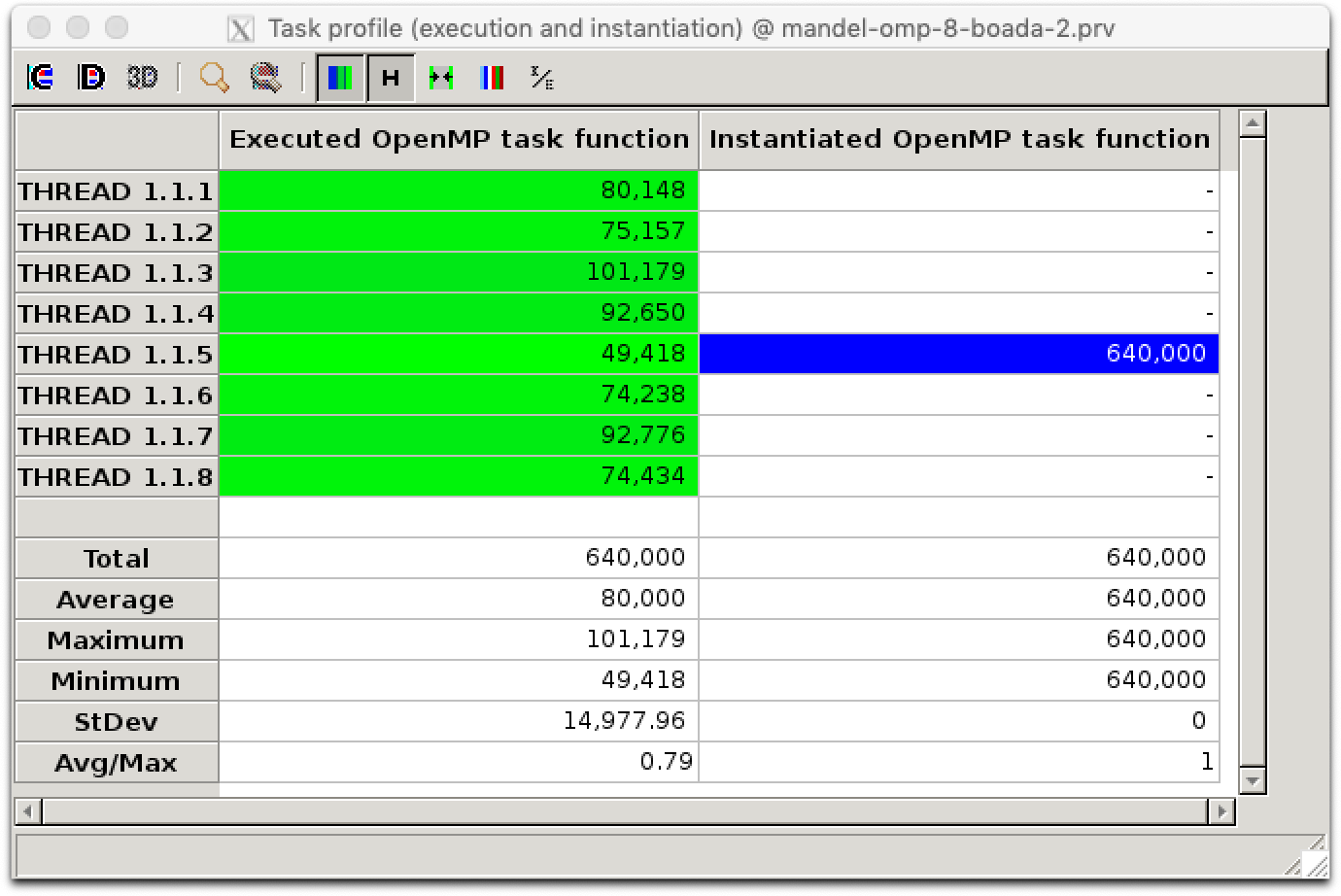
The *parallel* construct is invoked one time. The *single* worksharing construct is invoked one time. The *taskwait* construct is invoked 800 times (the value of the height variable).

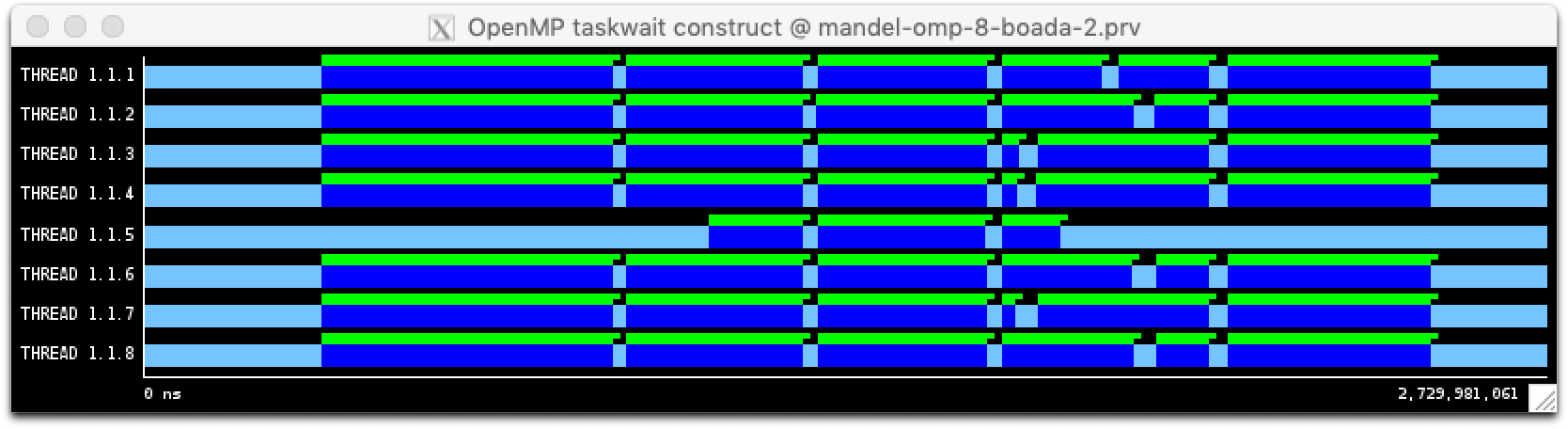
640.000 tasks are being created and executed.

The size and the number of the tasks is the same as the previous versions, the main difference is that in the second all the tasks are created by the same thread. So the granularity doesn’t change.

Taskwait is not necessary, because there are no dependencies between tasks.



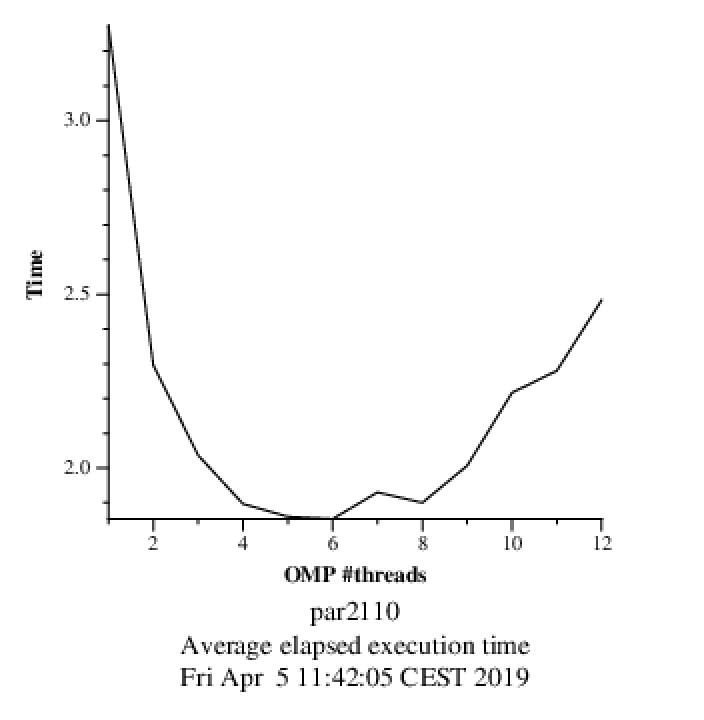


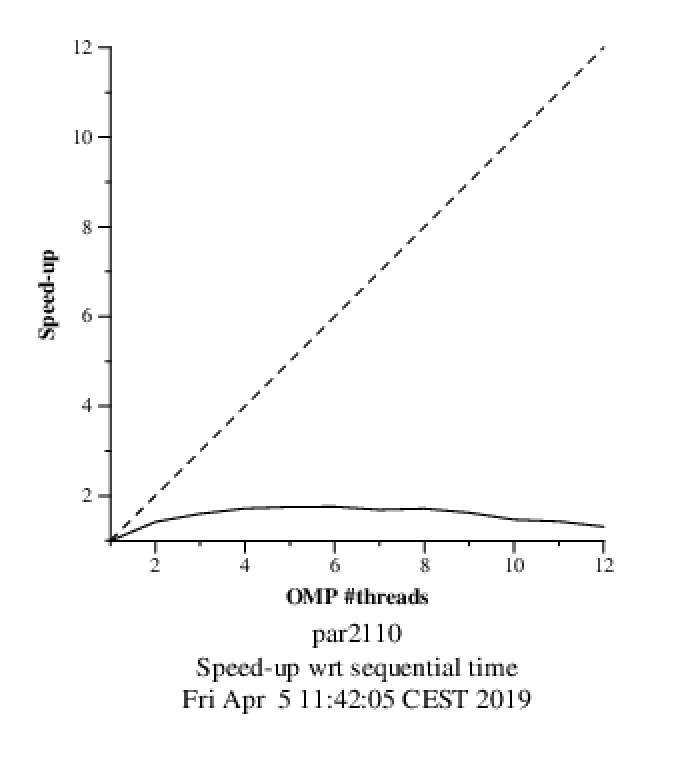


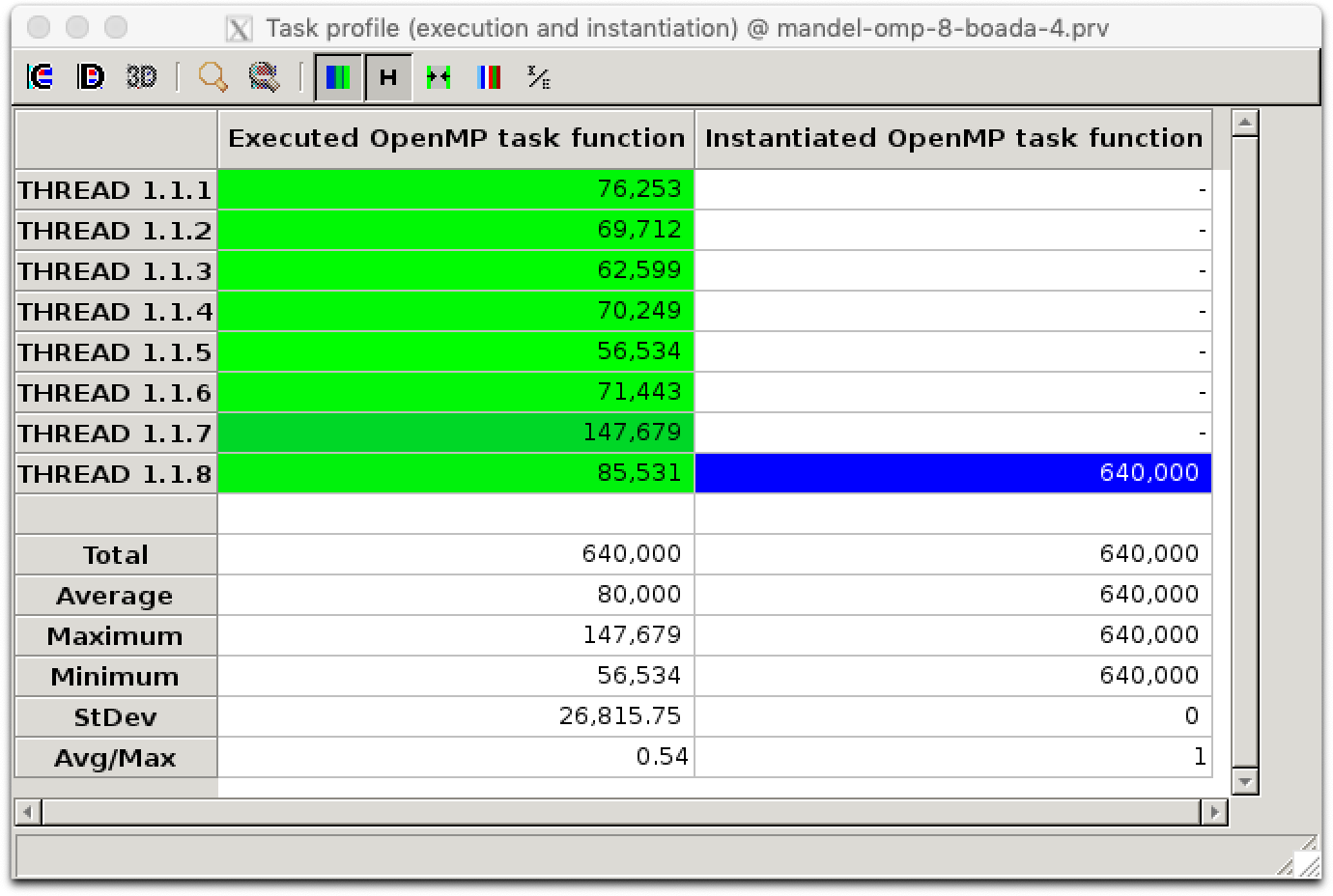
## 

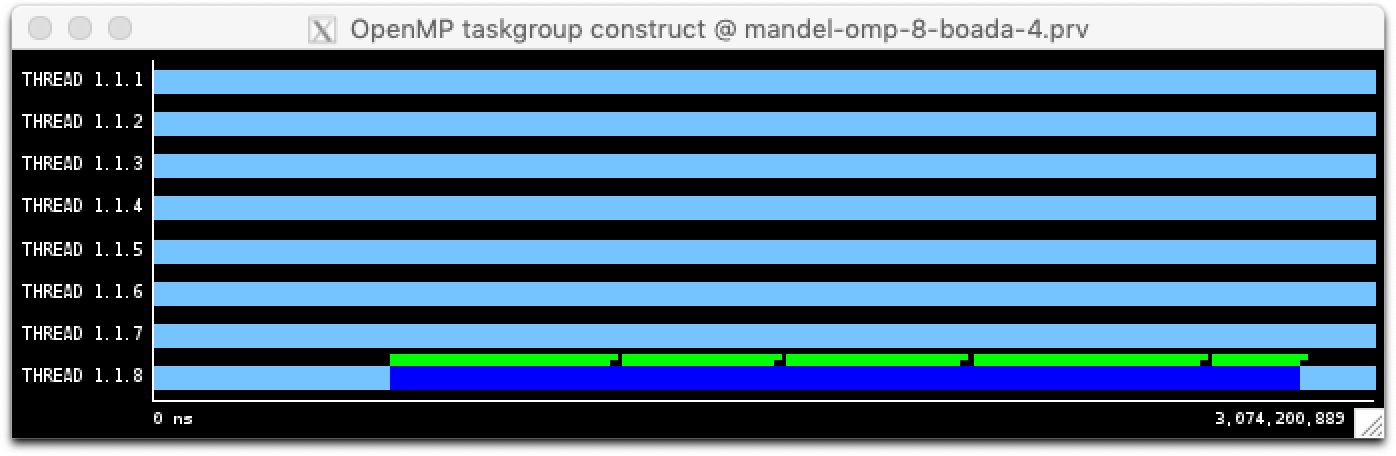
## 3. Version (Taskgroup)

|  |
| --- |
| #pragma omp parallel  #pragma omp single  for (row = 0; row < height; ++row) {   #pragma omp taskgroup  {   for (col = 0; col < width; ++col) {  #pragma omp task firstprivate(row, col)  {  ... |





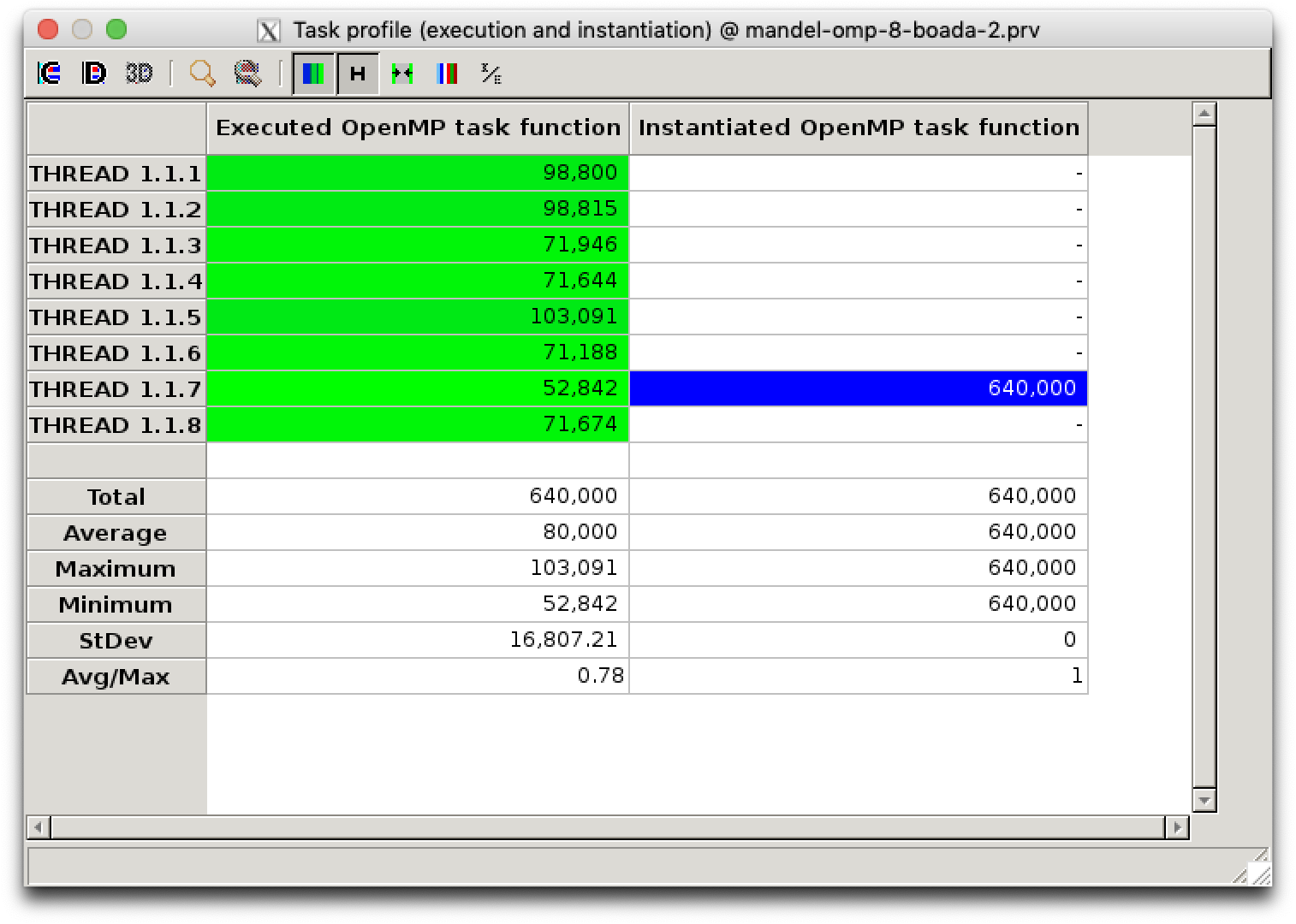


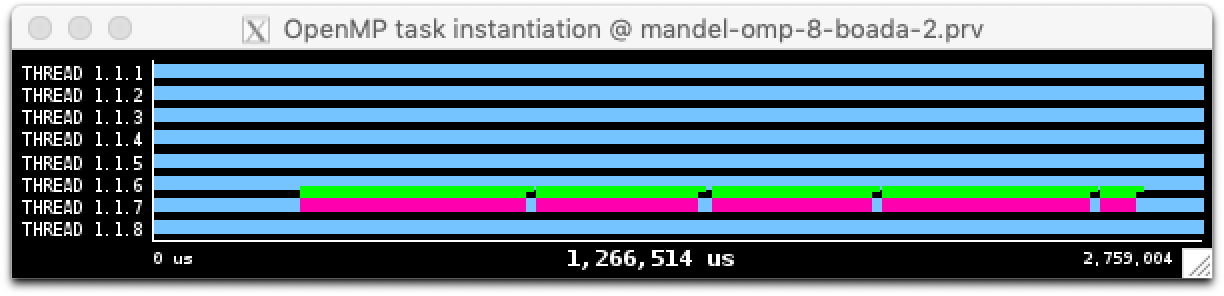


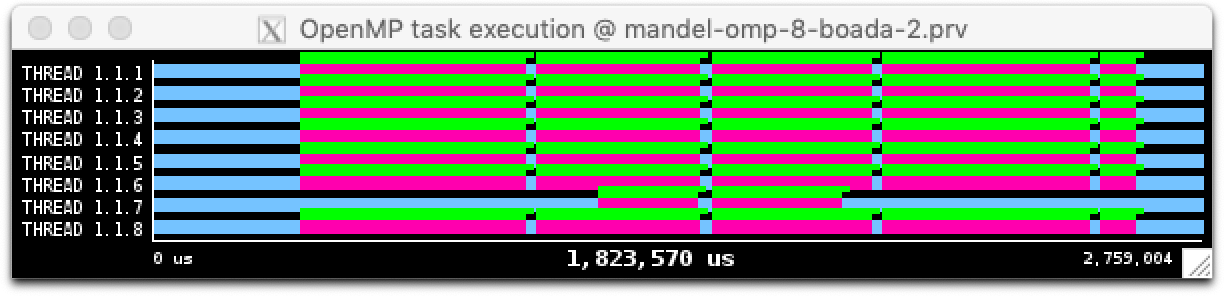
## 

## 4. Version (Task without Taskwait)

|  |
| --- |
| #pragma omp parallel  #pragma omp single  for (row = 0; row < height; ++row) {  for (col = 0; col < width; ++col) {  #pragma omp task firstprivate(row, col)  {  ... |



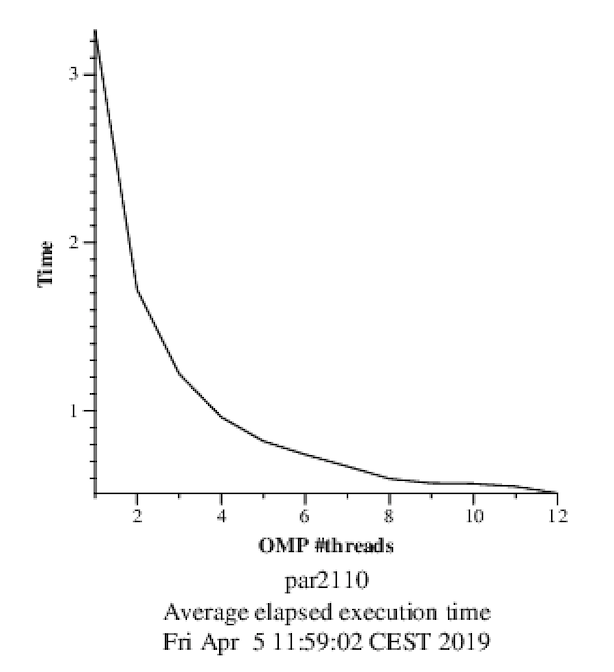


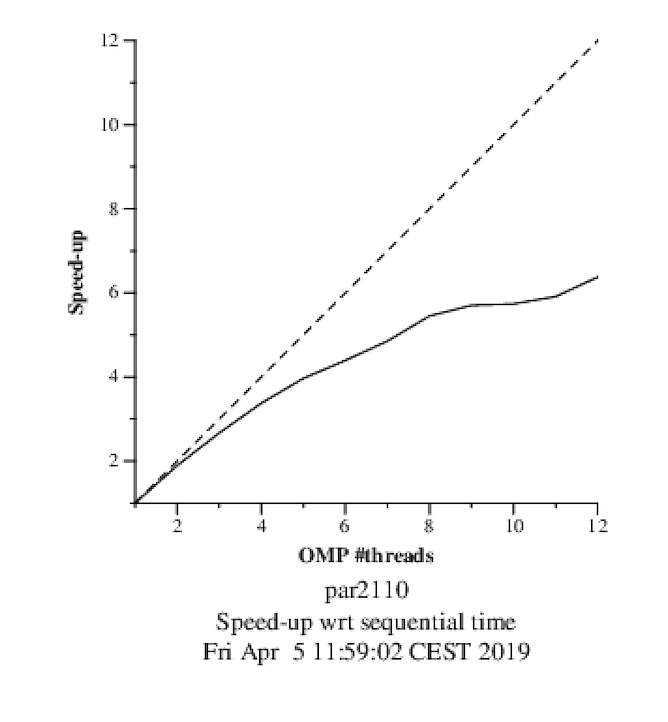


It stops creating tasks because it reaches the maximum, then it starts executing and creating, until tasks are finished faster than created.

## 5. Version (Taskloop with Grain Control)

|  |
| --- |
| #pragma omp parallel #pragma omp single for (row = 0; row < height; ++row) {   #pragma omp taskloop firstprivate(row) num\_tasks(800)   for (col = 0; col < width; ++col) {   ... |





It’s better because there is less overhead due to the number of tasks created.

We can’t delete the implicit taskloop barrier if we use the display version, but we could if we didn’t.

### Different Grain SizesGráfico

|  |  |
| --- | --- |
| Num. Tasks | Seconds |
| 800 | 0.176430s |
| 400 | 0.227919s |
| 100 | 0.169144s |
| 50 | 0.084223s |
| 25 | 0.057616s |
| 10 | 0.036864s |
| 5 | 0.031803s |
| 2 | 0.029964s |
| 1 | 0.029803s |

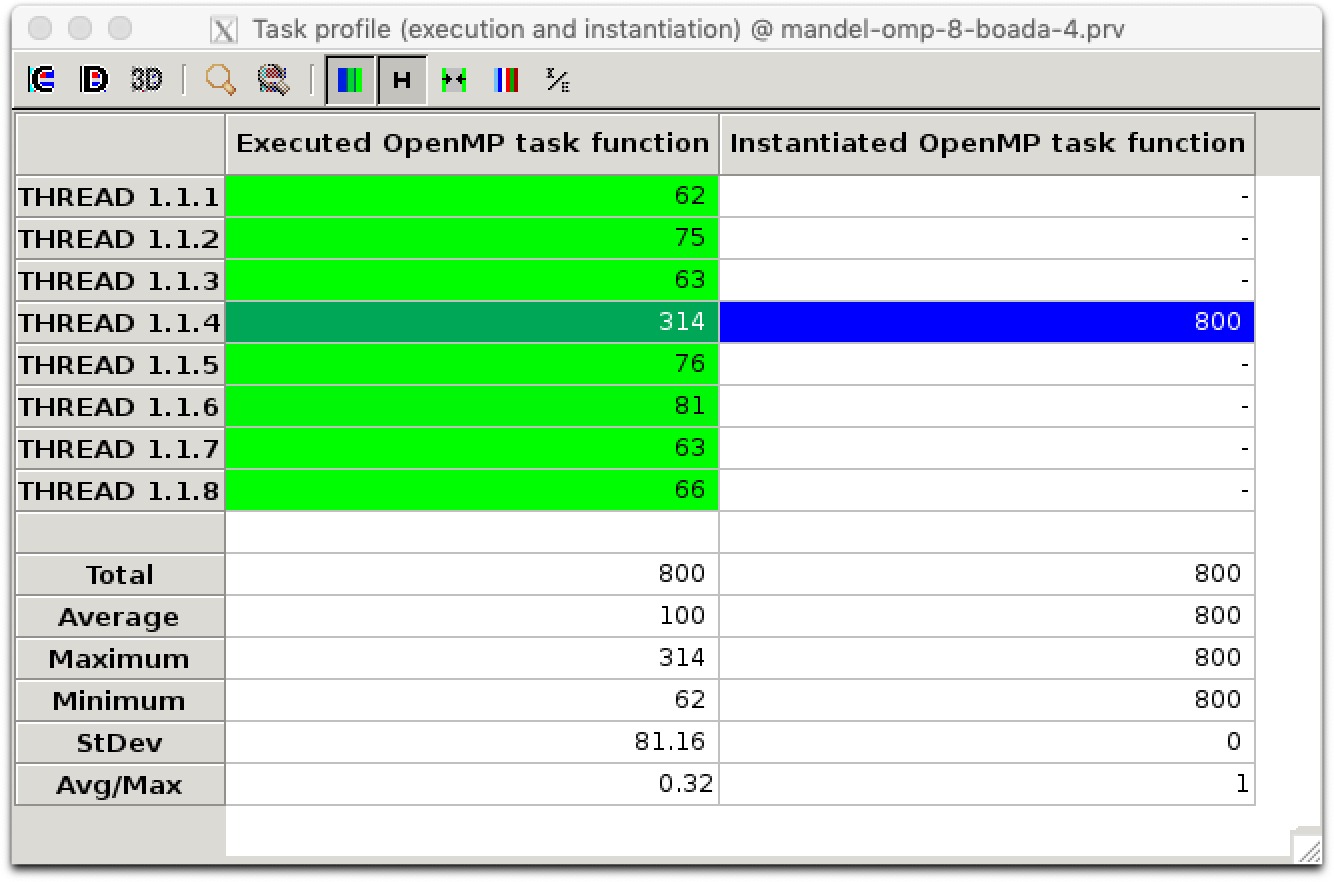
This results show that is takes more time until, and then it stays roughly the same.

# Row decomposition in OpenMP

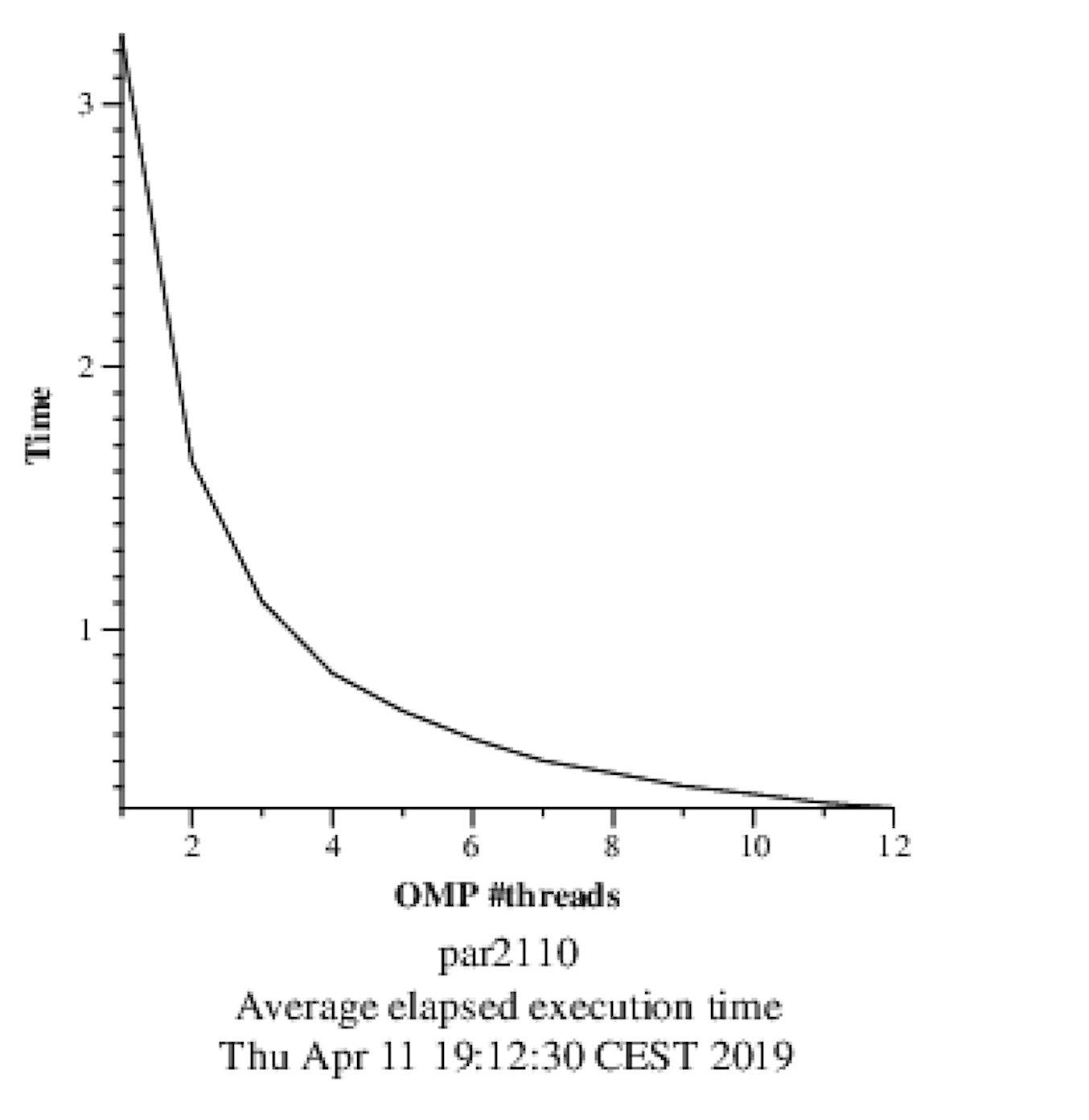
The following code was used for the row decompositions

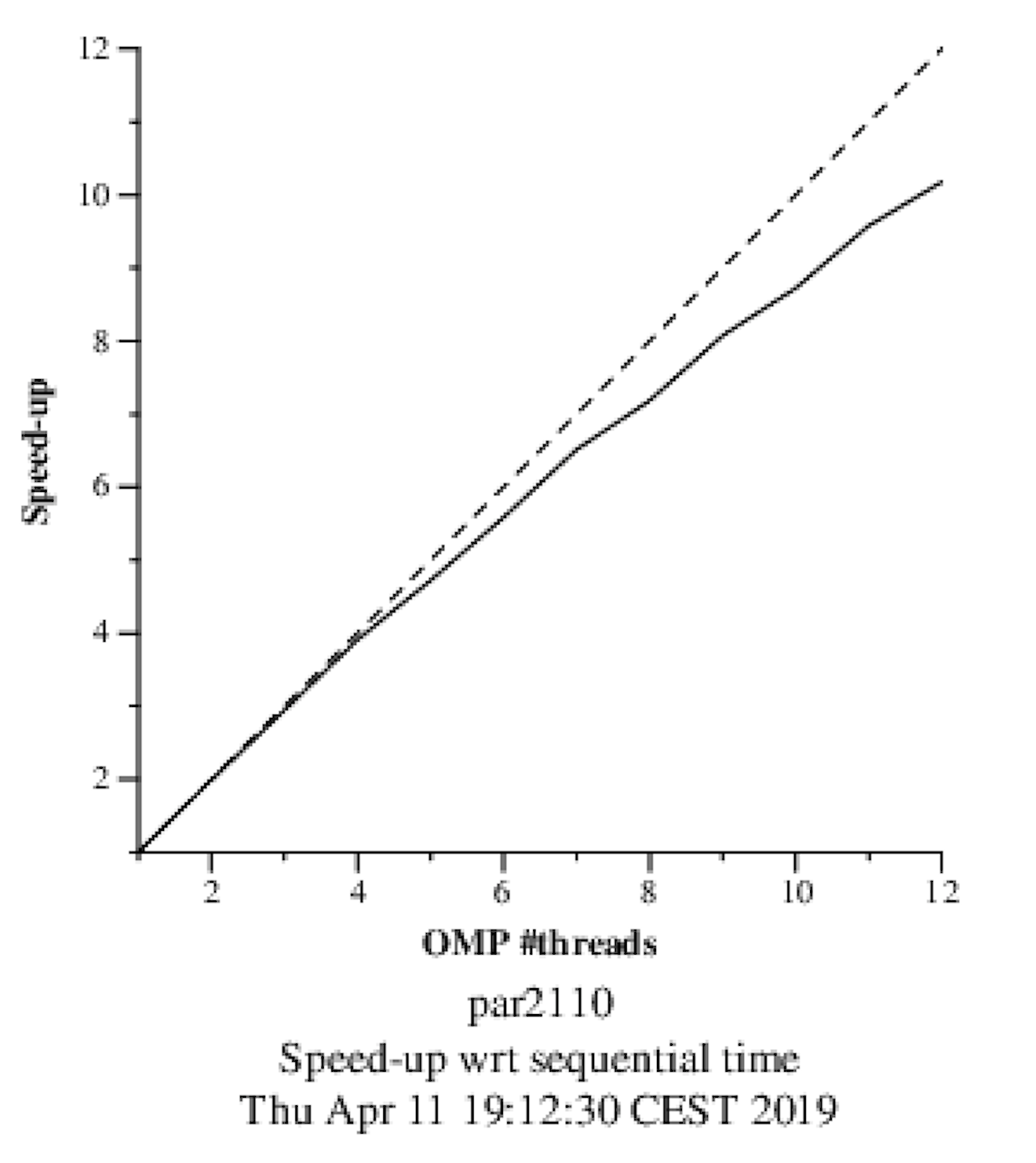
|  |
| --- |
| /\* Calculate points and save/display \*/  #pragma omp parallel  #pragma omp single  for (int row = 0; row < height; ++row) {  #pragma omp task firstprivate(row)   for (int col = 0; col < width; ++col) {    complex z, c; |

The Task execution profile shows less tasks in total



The Speed-up per threads is the following:



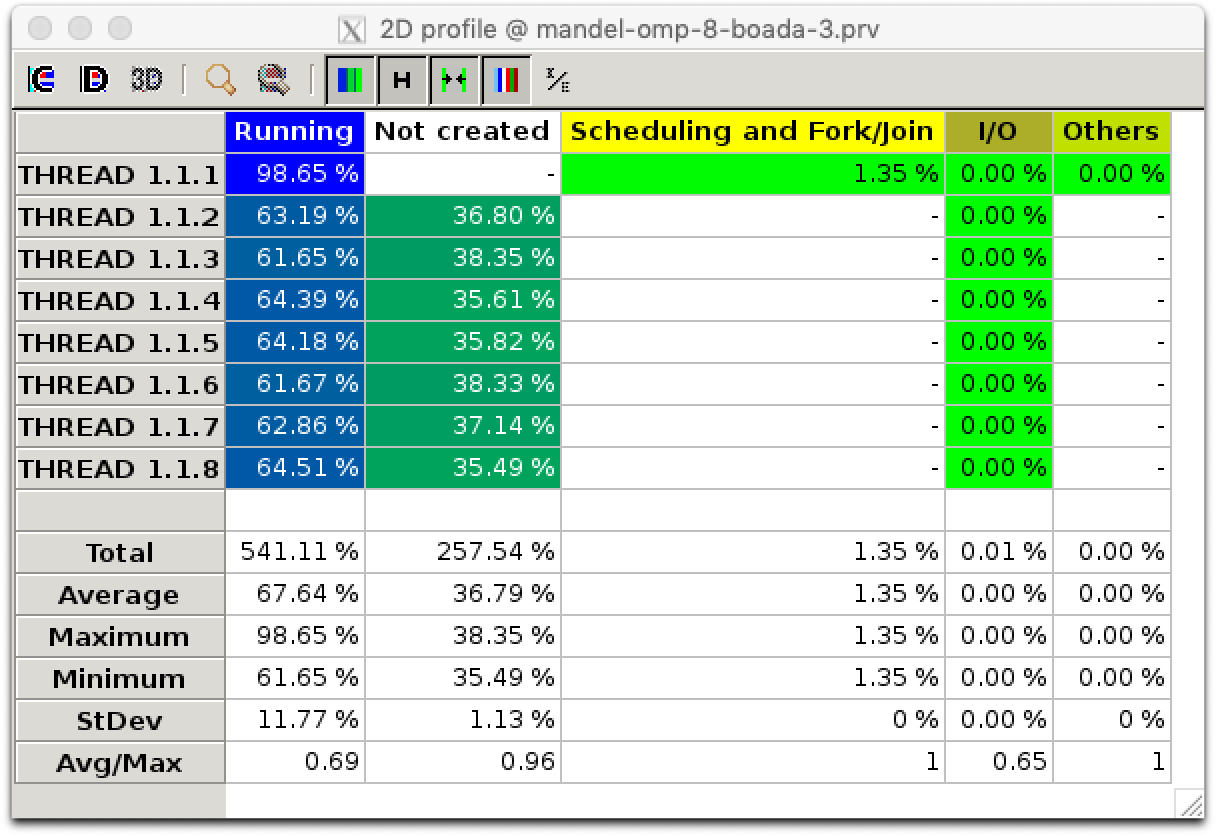


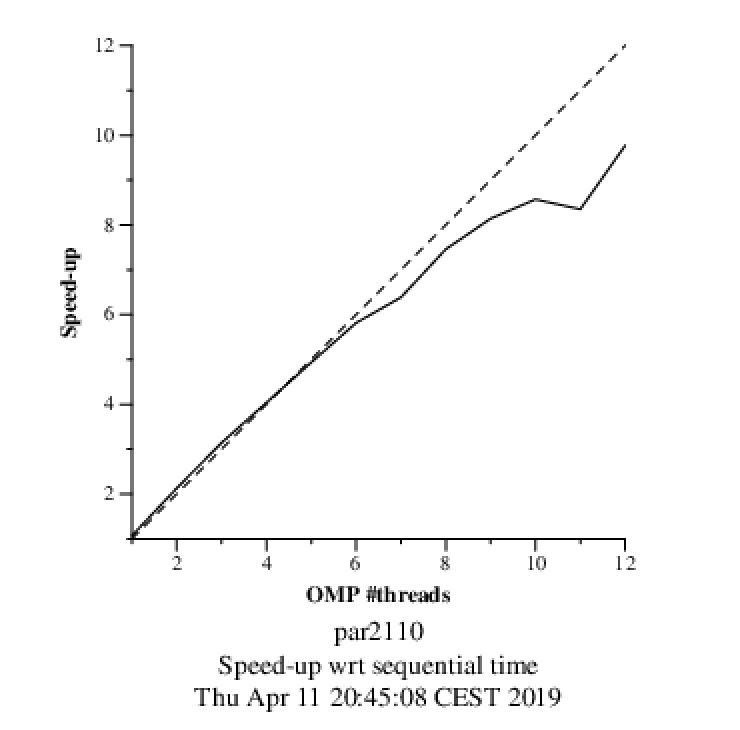
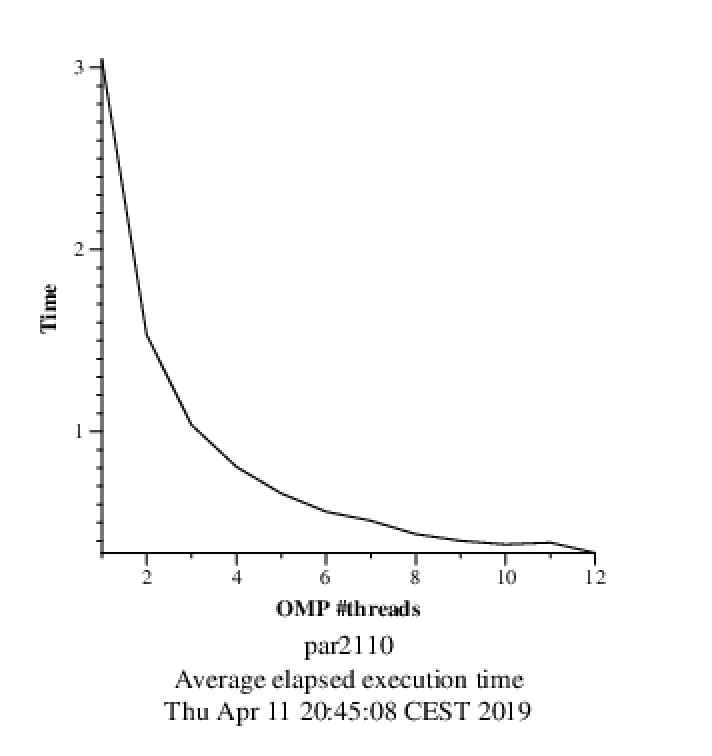
This version scales very well.

# For–based parallelization

|  |
| --- |
| /\* Calculate points and save/display \*/  #pragma omp parallel for schedule(static,10)  for (int row = 0; row < height; ++row) {  for (int col = 0; col < width; ++col) {   complex z, c; |

Total execution time is 0.033910s.





This version is slightly faster, but scales slightly worse. The first thread does more work than the others.