PARAL·LELISME: LABORATORI 3

# 

Pol Casacuberta Gil i Salvador Grimalt BInimelis

Quadrimestre 1, curs 22-23

Paral·lelisme, grup 21

[**Introduction**](#_8duo60g1uj7h) **3**

[**2 Task decomposition analysis for the Mandelbrot set computation**](#_qpeiyt99cw2x) **4**

[2.1 The Mandelbrot set](#_mr7x5meha87g) 4

[2.2 Task decomposition analysis with Tareador](#_x7a31bjbpgn6) 4

[Row strategy](#_hh6x225sva6) 4

[Point Strategy](#_uvhakyvv7cbg) 6

[**3 Implementation and analysis of task decompositions in OpenMP**](#_sghxow11xqrl) **7**

[3.1 Point decomposition strategy](#_8g5bcp593nv1) 7

[Using taskloop](#_2a4zhjsh124f) 11

[Taskloop nogroup](#_38qk97piykap) 15

[3.2 Row decomposition strategy](#_8h3bkv9ps0wo) 17

[3.3 Optional: task granularity tune](#_z03x8513wzf) 21

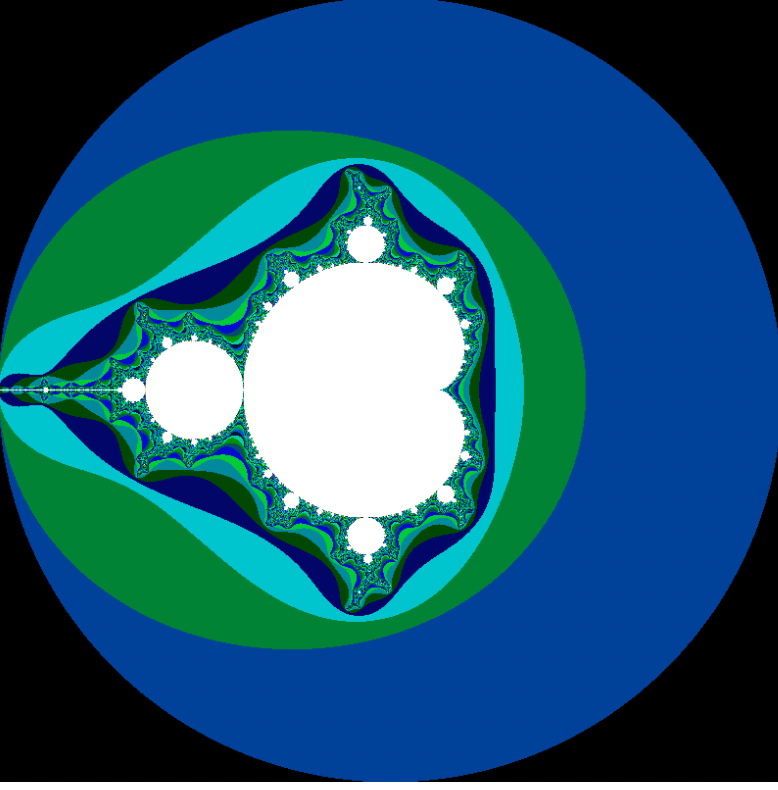
[Conclusions](#_eh3qeq588s59) **24**

# Introduction

The main goal of this project is to investigate the different ways to parallelize the given program which is used to compute the Mandelbrot set. We investigate the point-focused strategy and a row-focused strategy. This will help us understand the difference between these ways to parallelize the same program.

The Mandelbrot set is a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognizable two-dimensional fractal shape. The way the program processes the set is by choosing a section of this set and iteration over all the points of this section to see to which “part” they belong to. With the given piece of code we can do several things as displaying a graphical representation of the mandelbrot set, we can also get an output and a histogram of the set.

The default image should look a lot like this one:



# 2 Task decomposition analysis for the Mandelbrot set computation

## 2.1 The Mandelbrot set

## 2.2 Task decomposition analysis with Tareador

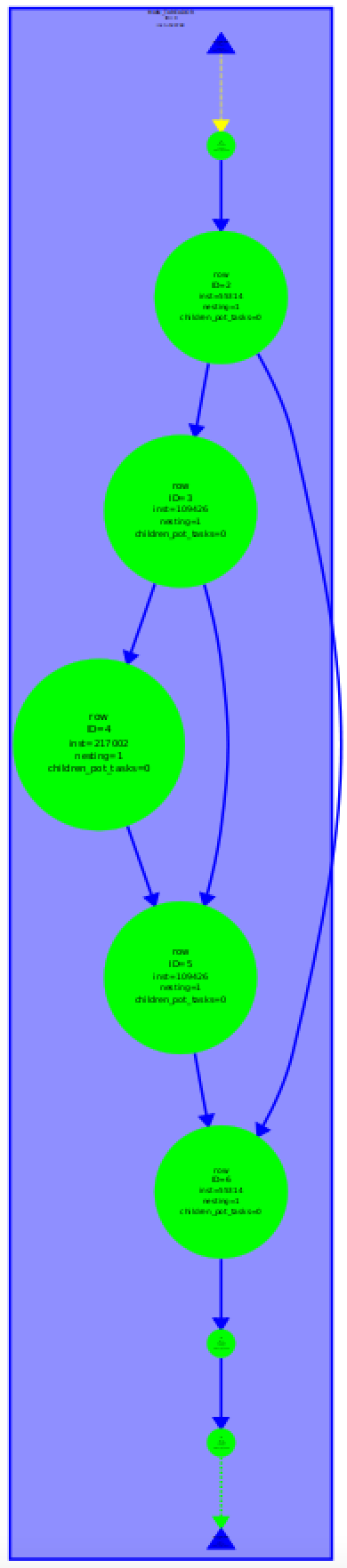
## Row strategy

# 

The first thing that we notice with this code it that it is completely parallel, maybe even “embarrassingly parallel”. Another thing to notice is that workload doesn’t seem to be equally distributed between tasks. Some tasks are tiny, whilst some tasks are huge in comparison. That means task granularity is not distributed equally amongst tasks, which is not a good thing. Because some tasks will take a lot longer than others, and that could end up being a bottleneck.

On the next piece of code we are asked to execute with the flag -d which displays a graphical representation of the mandelbrot set. We now see that the code has become completely sequential, with then again, tasks that are tiny and tasks that are big. The piece of code that causes it, is the following:

| if (output2display) {  /\* 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);  }  } |
| --- |

One possible solution to protecting this piece of code would be to add pragma omp critical to the entire inner if. This is because of how XSetForeground and XDrawPoint work internally, they must be executed one immediately after the other.

As far as the mandel-tar binary run with the -h option we find that we get a sequential execution of our code as last time, but now we have two more dependencies. The following piece of code is the reason our code is not executing in parallel.

| if (output2histogram) { histogram[k-1]++; } |
| --- |

To protect this piece of code we could simply use a pragma omp atomic so that it would not have data races and could be executed in parallel.

# 

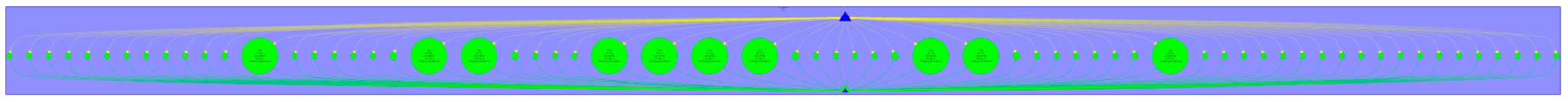
# 

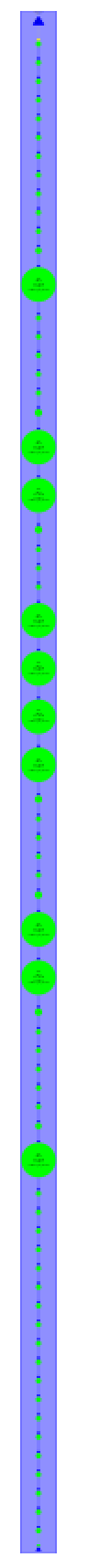
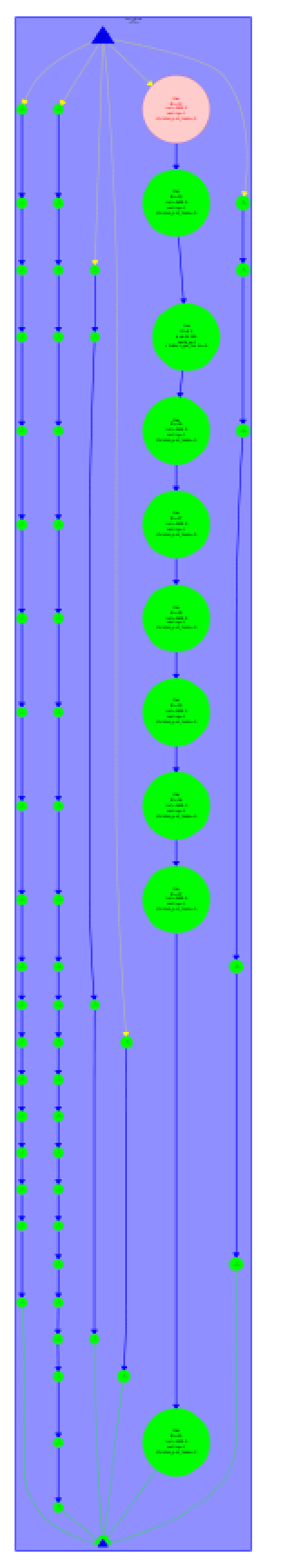
# 

# 

# 

## Point Strategy





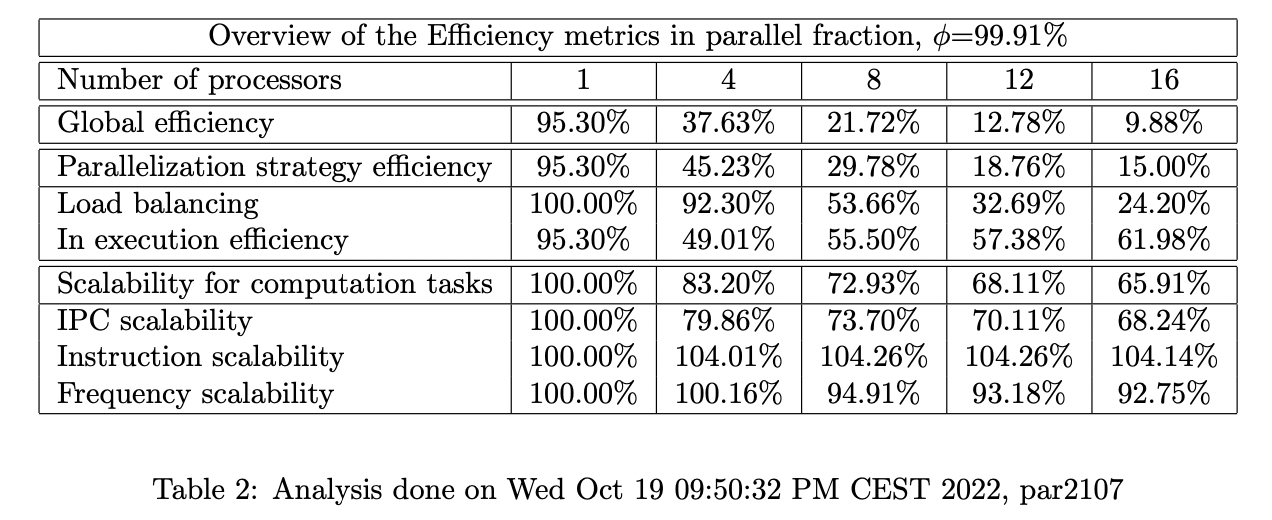
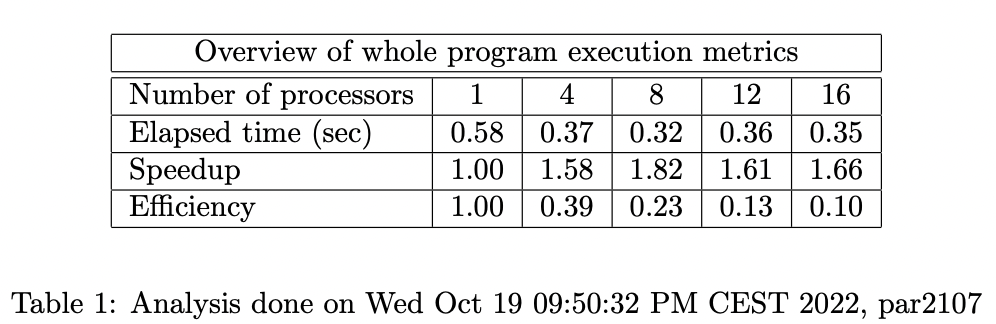
Here we can see the potential parallelism that could be achieved with Point strategy. With mandel-tar we get an embarrassingly parallel execution and we see that some tasks are significantly bigger than the others. The grainsize of these tasks is much finer than the ones with row strategy.

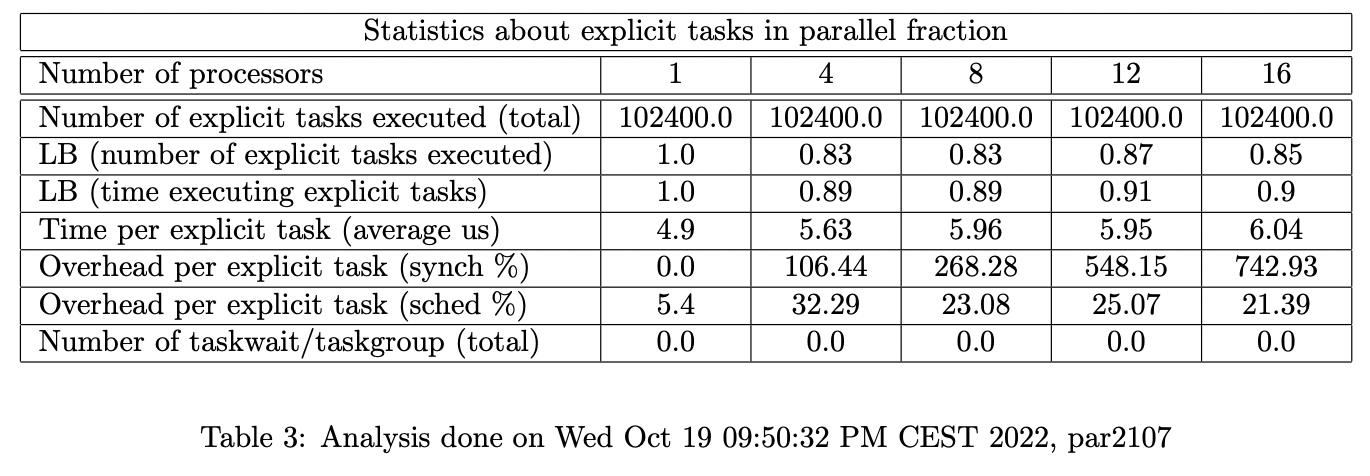
Next up we can see the execution of mandel-tar -d which is entirely sequential and mandel-tar -h which is somewhat parallel but mostly sequential.

Mainly the difference between Point and Row strategy is the much finer grain, could be worth it depending on the overheads that could be added with point strategy and on the number of processors that we have available.

# 3 Implementation and analysis of task decompositions in OpenMP

## 3.1 Point decomposition strategy





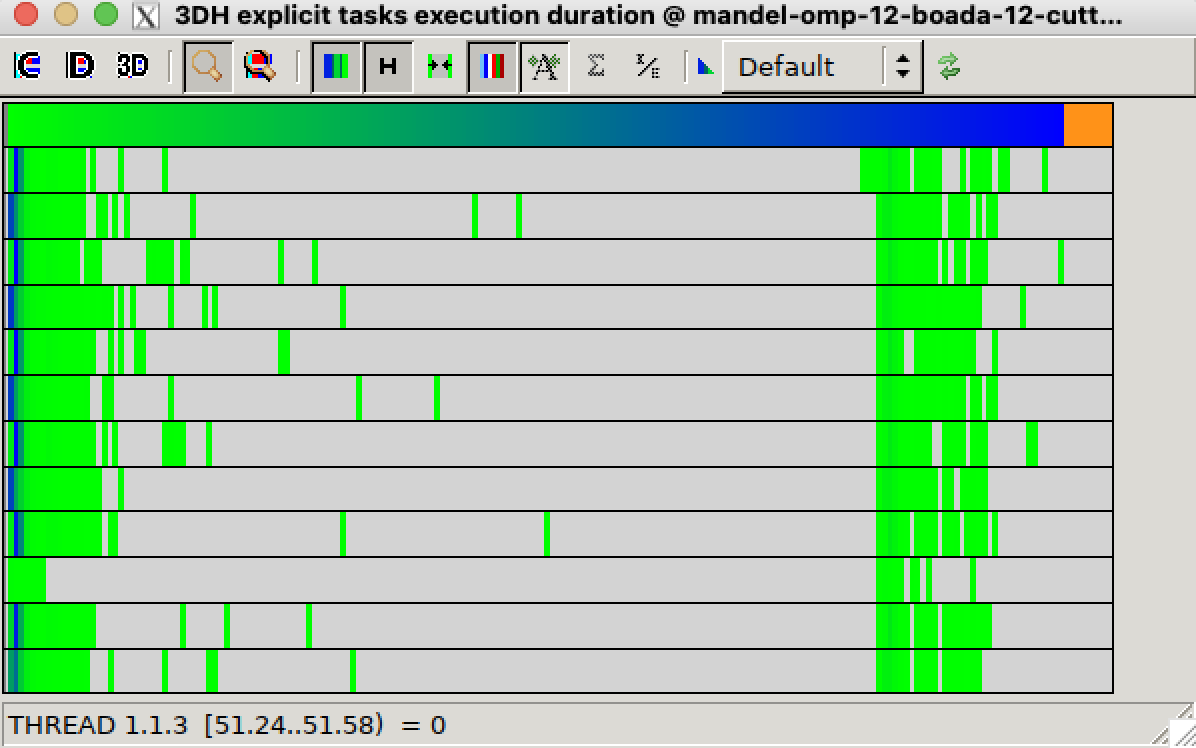
First of all we performed an analysis of the Point strategy using the submit-strong-extrae.sh which performs Extrae instrumented executions of the code for 1,4,8,12 and 16 processors.

If we take a look at the speedup we see that it caps out at 8 processors and then at 12 and 16 performance goes down, probably because of overheads.

Scalability seems to be pretty bad as well, since it decreases pretty rapidly. It also looks like load balancing is pretty awful as well, and could be causing big performance hits.

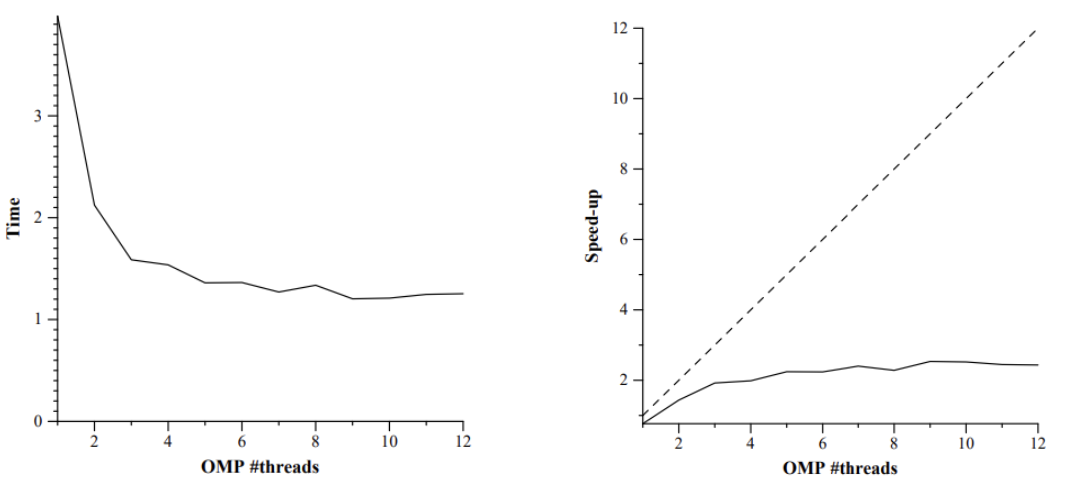
## 

There is a single thread that is creating the tasks, and that same thread is executing much less explicit tasks, this should be our source of the problem for our load unbalance. The rest of the threads are all working all the time on those explicit tasks.



We see that the explicit tasks take a big chunk of time they are pretty sparse.

The granularization in this version of code seems inadequate, perhaps because it is too coarse, there are too many overheads that are causing that the code performs, at best, suboptimally.



## 

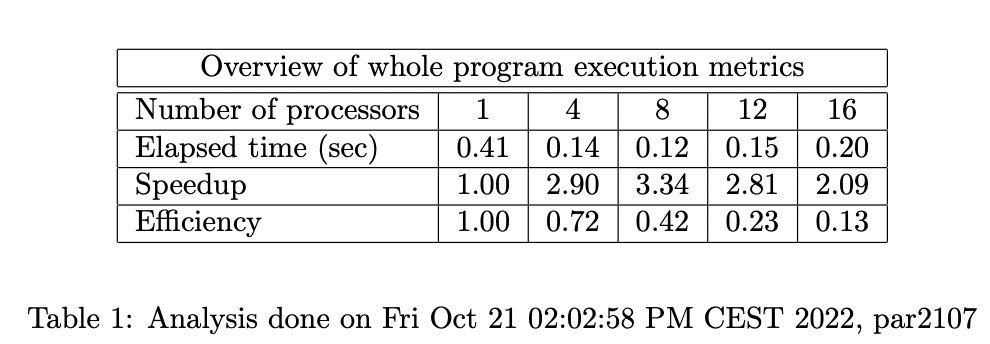
The number of tasks created by one thread is 102.400 and the total number of tasks executed by all threads is exactly that 102.400.

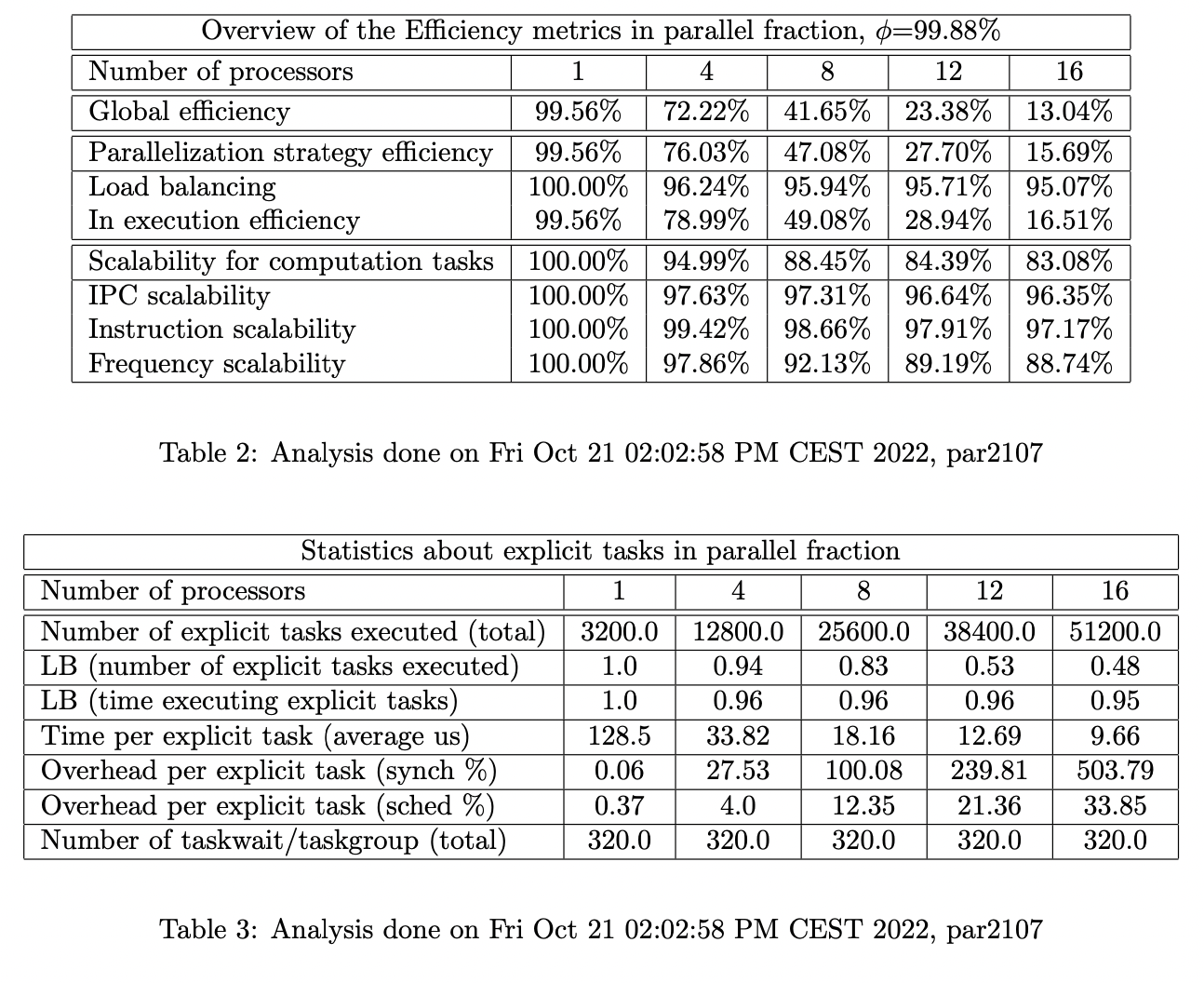
Here is the extract of the code used:

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

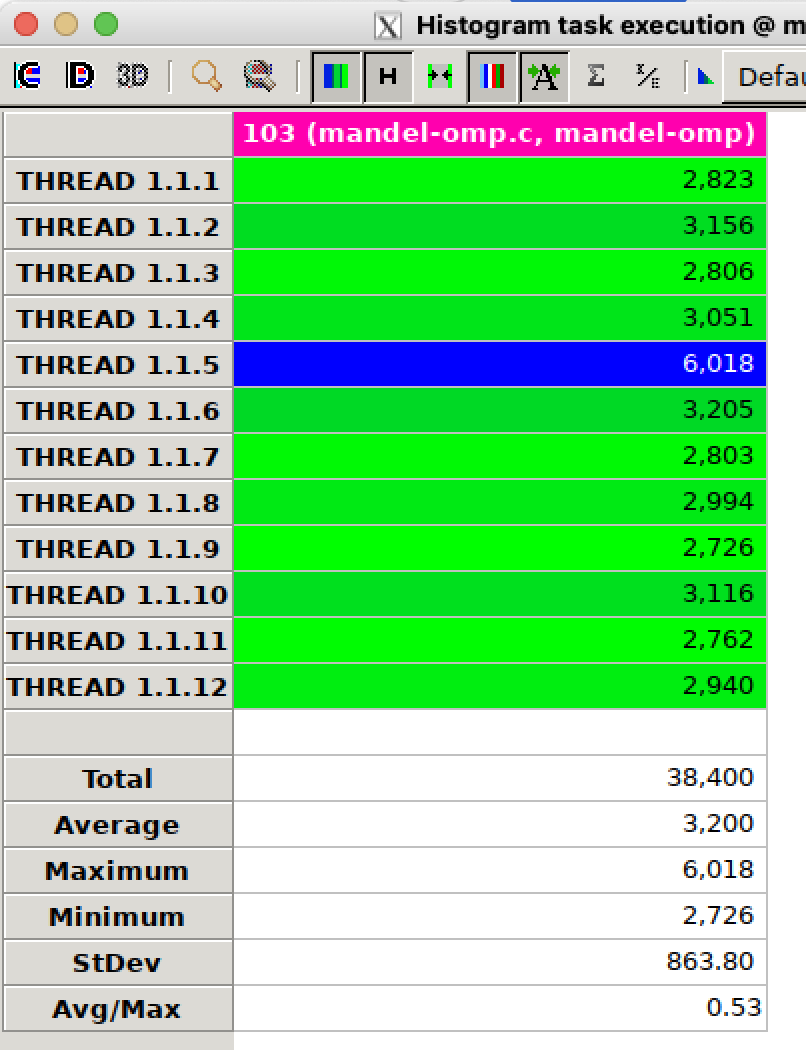
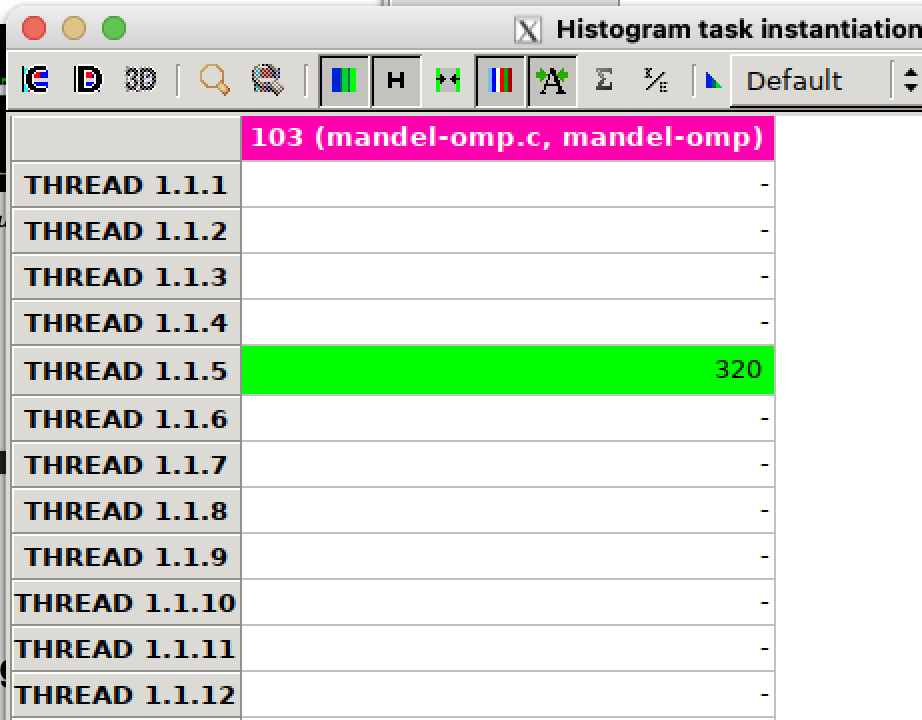
## Using taskloop

| #pragma omp parallel  #pragma omp single    for (int row = 0; row < height; ++row) {  #pragma omp taskloop  for (int col = 0; col < width; ++col) {  }  } |
| --- |





Performance with taskloop seems a lot better, but we still have scalability issues. Looking at the two metrics that contribute to parallelization strategy efficiency it seems that load balancing has improved dramatically compared to the previous version with tasks, it seems pretty clear that In execution efficiency seems to be at fault in this case, and we should try to remedy that.



320 tasks (number of rows) are created by the taskloop directive it seems to be different from the real number of tasks executed but this is because taskloop divides the iterations of the col loop in groups of tasks. Since there’s no granularity restriction, the code decides the number of tasks created and the number of loops on each task. Also the task granularity seems to be the same since the number of loop iterations is the same the granularity still remains the same. So we get 320 instantiated tasks and 38400 executed tasks this gives us 120 executed tasks per taskloop and each of the tasks have a grainsize of about 2.666 (320\*320/38400) iterations per task.

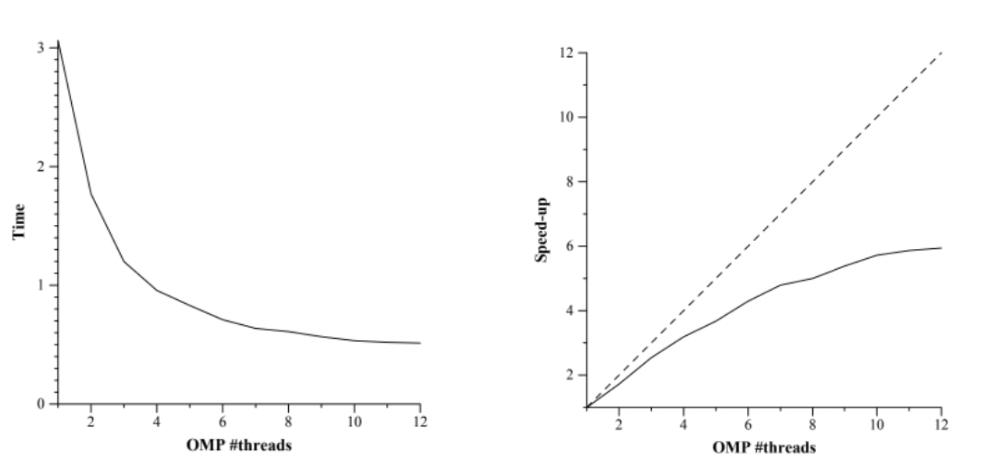
## 

Task synchronization is obviously still a major problem that we’ll need to solve. We can also see this in table 3, it is much lower than before, but it still is taking a big hit on performance.

## 

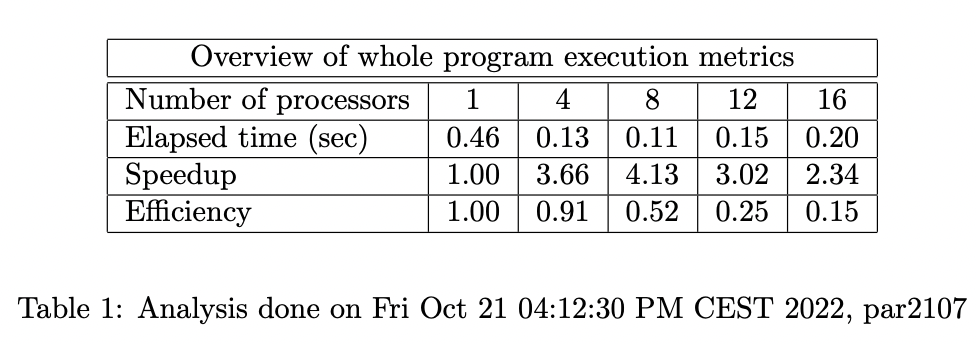
## 

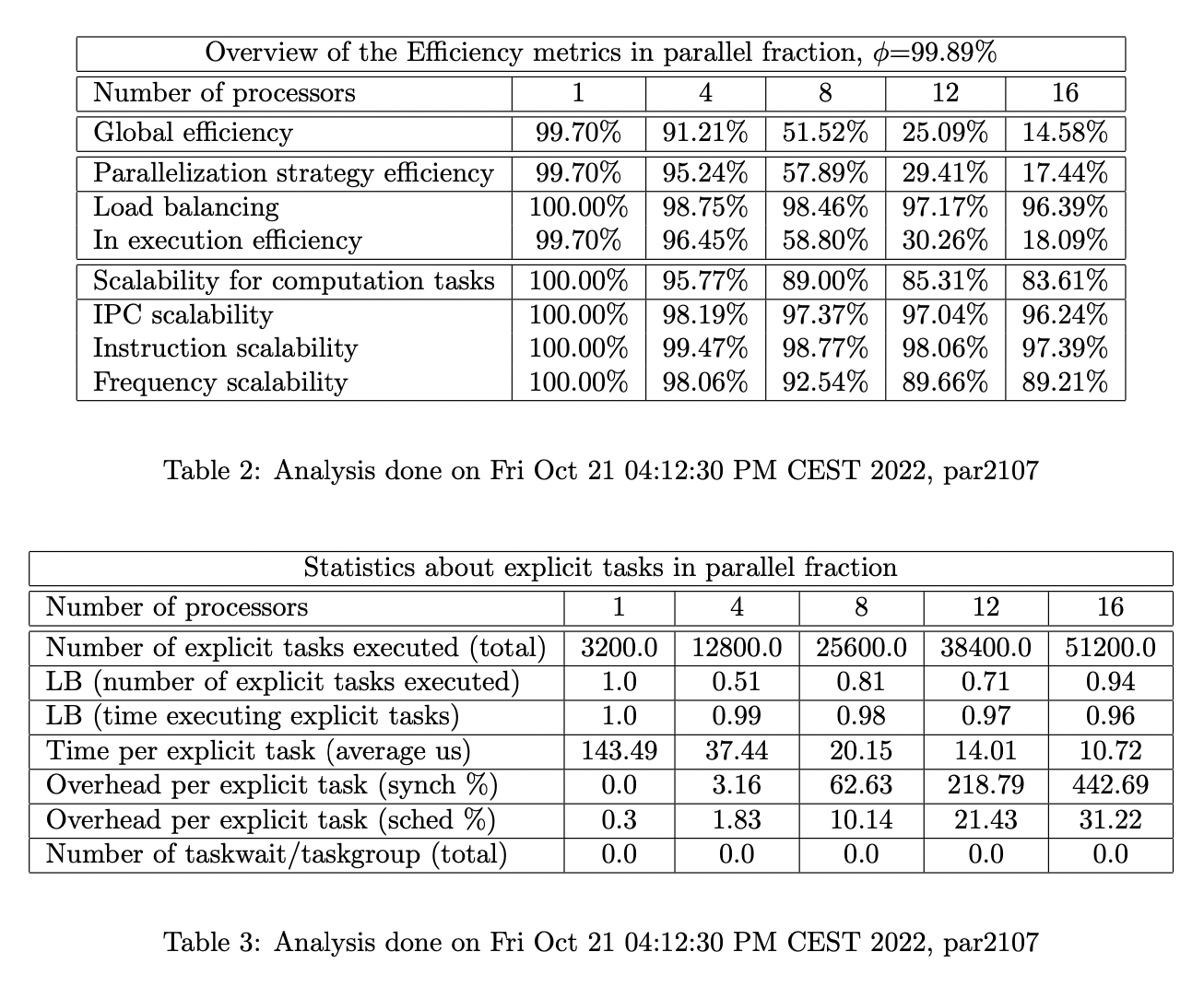
One big problem in performance could be that the tasks have to wait outside the taskloop construct every time before continuing to the next iteration. That seems to be where our task synchronization is happening. We could use taskloop with the “nogroup” directive to solve such problem.

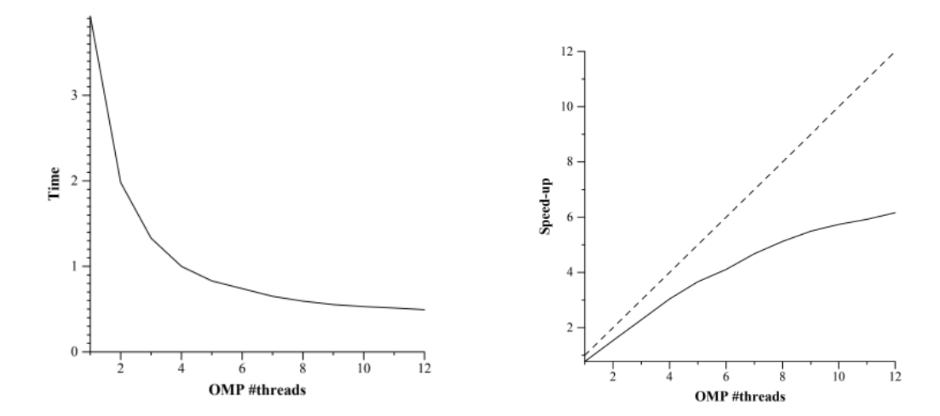


## Taskloop nogroup

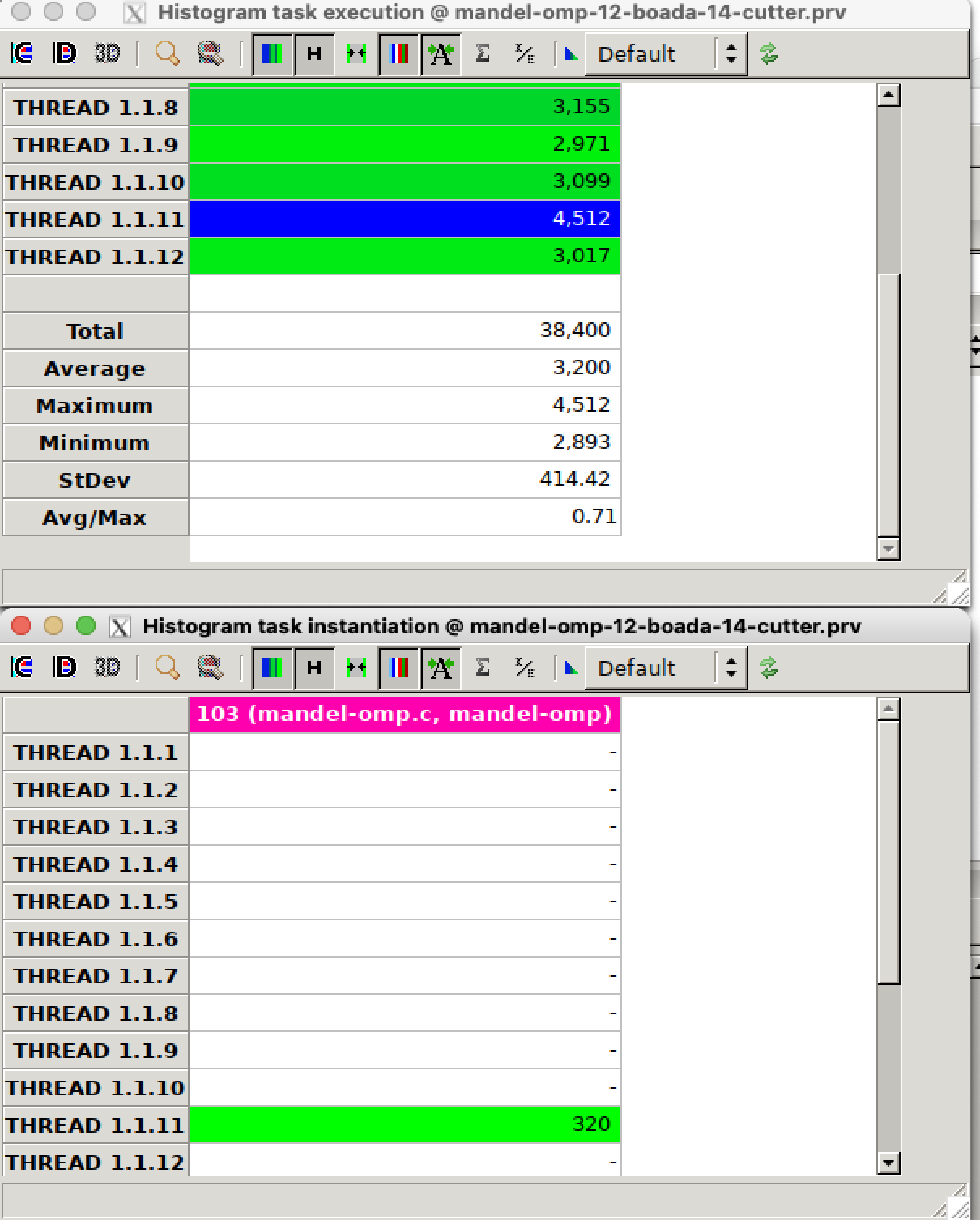
We added to taskloop the directive “firstprivate(row) nogroup“ to make sure multiple threads do not modify that variable. Earlier we didn’t need it because taskloop created an implicit taskgroup and waited at the end of the iteration of the inner loop, but that is not the case anymore.





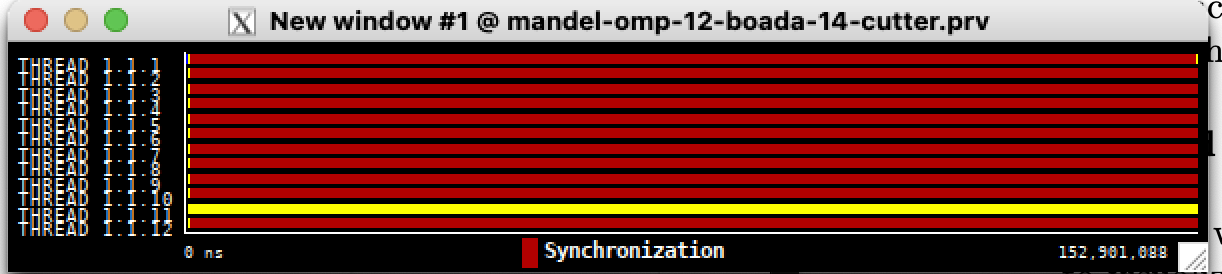


The performance of this version of code does seem to be a bit better in terms of the overhead due to synchronization, but there’s still much to be improved here. Scalability is better, but it is not significative.



The number of explicit tasks remains the same and the granularity also remains the same.

The results of all the code versions were verified both visually using mandel-omp -d and comparing with the diff command with the sequential version which we know to be correct.



Extract of the code used:

| #pragma omp parallel #pragma omp single for (int row = 0; row < height; ++row) {   #pragma omp taskloop firstprivate(row) nogroup   for (int col = 0; col < width; ++col) {   }  } |
| --- |

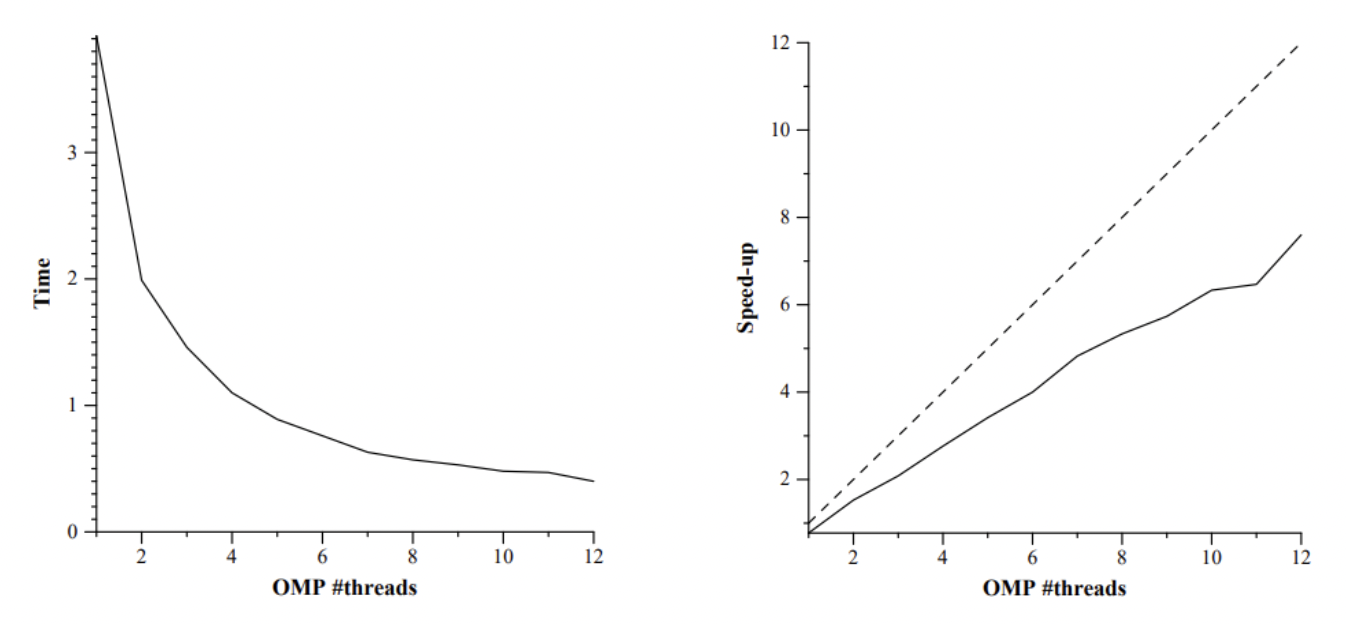
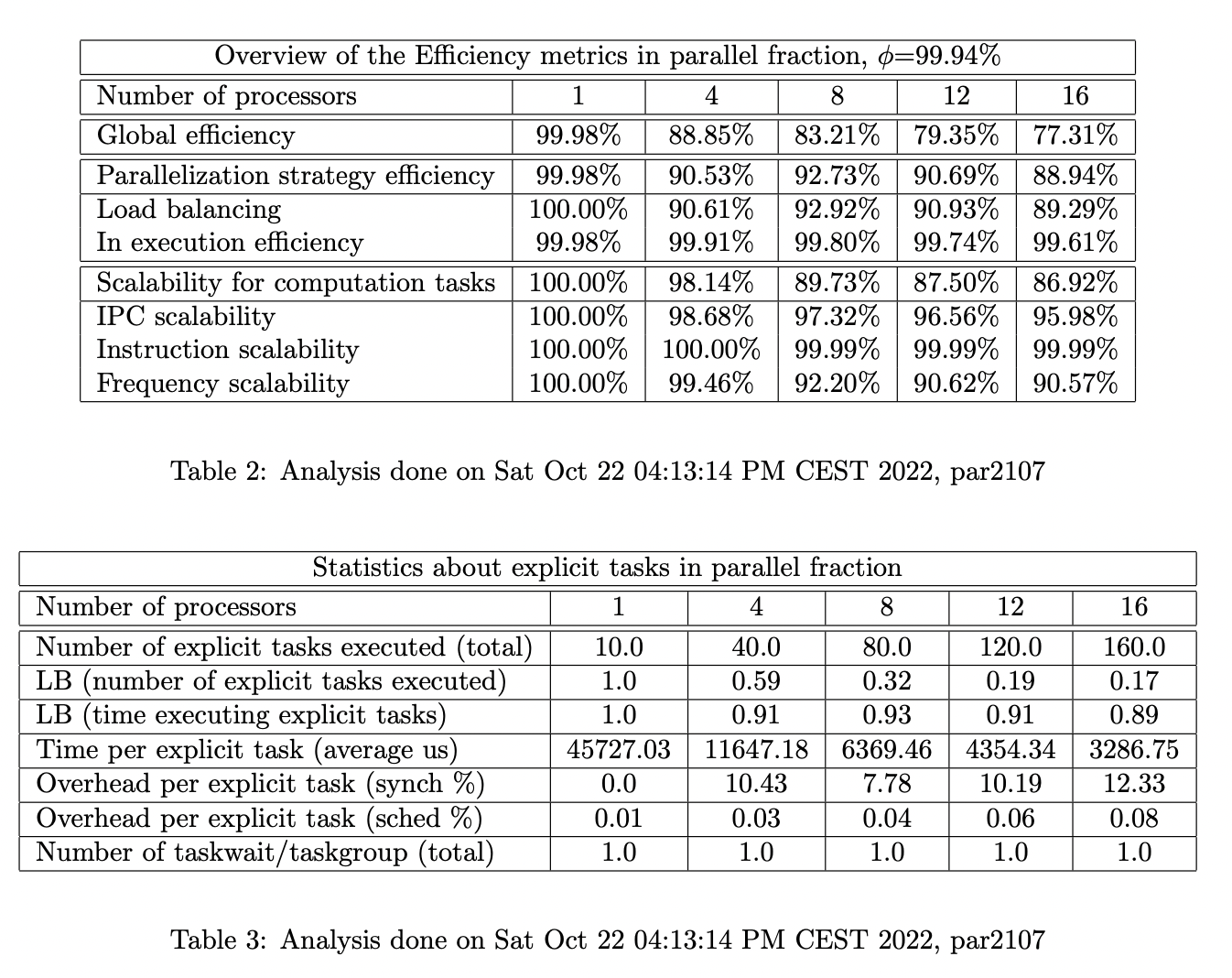
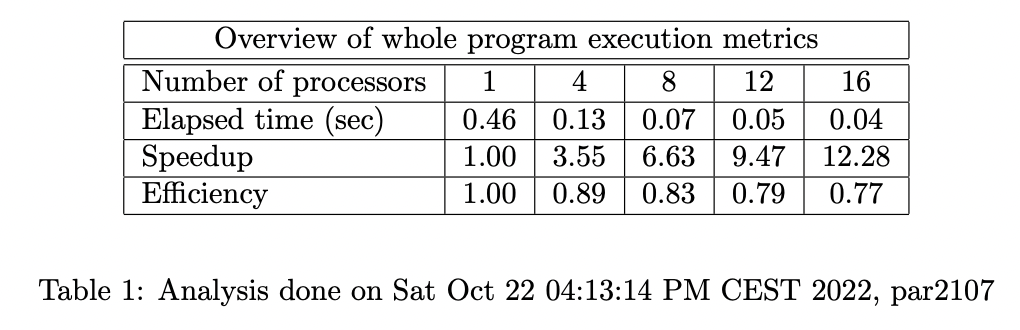
## 3.2 Row decomposition strategy

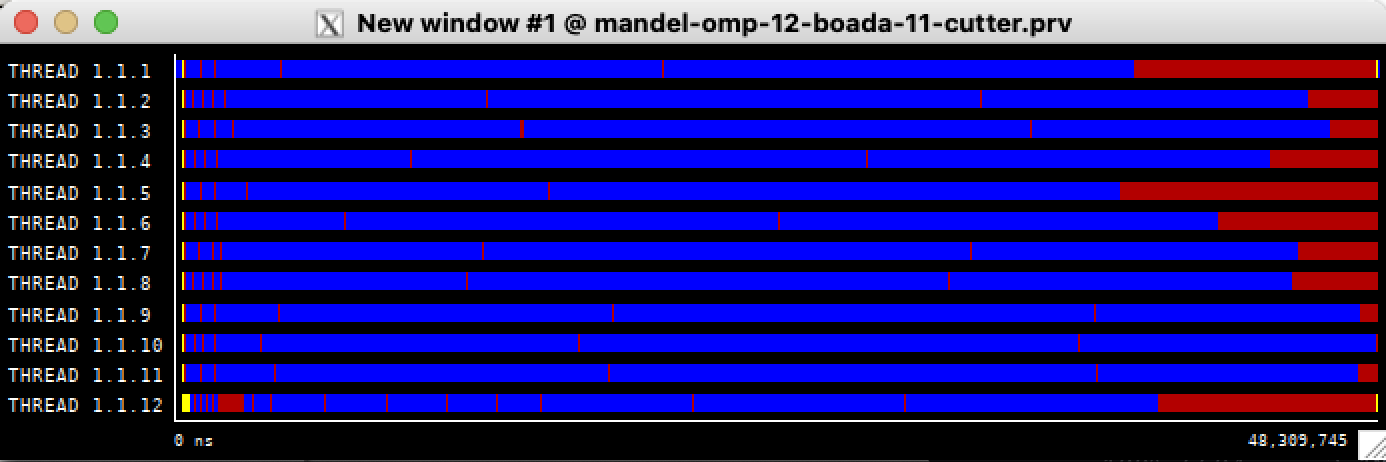
| #pragma omp parallel  #pragma omp single  #pragma omp taskloop  for (int row = 0; row < height; ++row) {  for (int col = 0; col < width; ++col) {  }  } |
| --- |

In this version of the code we adapted the code we had so that it would create a task for each row, this means that each task will be getting a similar number of rows to process, which means load balancing should be pretty good.

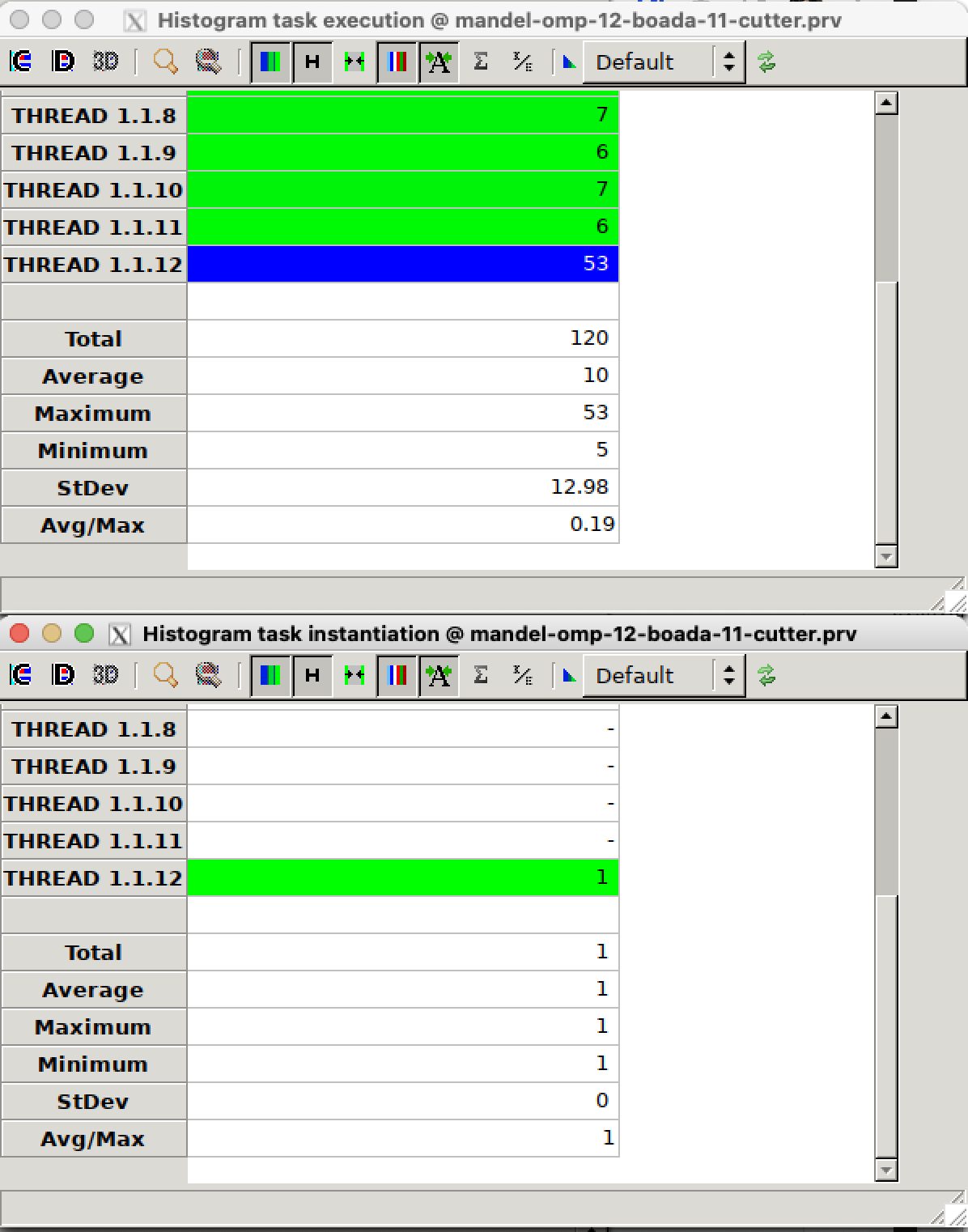
We checked that the result was correct by checking out the output images, and comparing the outputs with the diff command.

The speedup in this version of the code seems to be miles better than the one we had on the previous version, and it seems the speedup keeps increasing even when using 16 processors. This probably means that we solved major overheads.

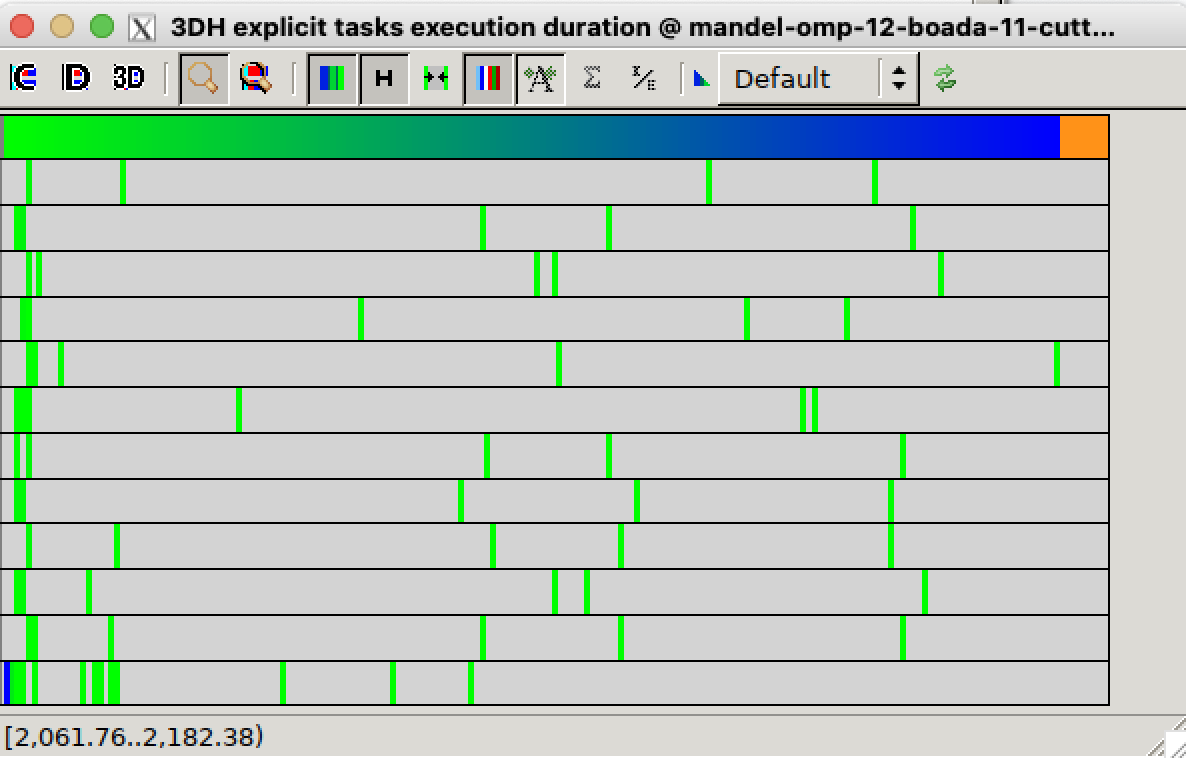




We can see here that the threads are mostly running all the time, in contrast to what we had before where they were mostly doing synchronization tasks.



We can see here that there’s only one creation of a task and the execution of 120 tasks this is similar to what we’ve already seen when using taskloop with the point strategy. We have 102400 iterations and 120 tasks so the granularity that we have corresponds to 853.3 iterations/task.



We can also see in this explicit task duration histogram that it is much more disperse.

## 3.3 Optional: task granularity tune

Here are the codes that were used to extract the results:

Point strategy

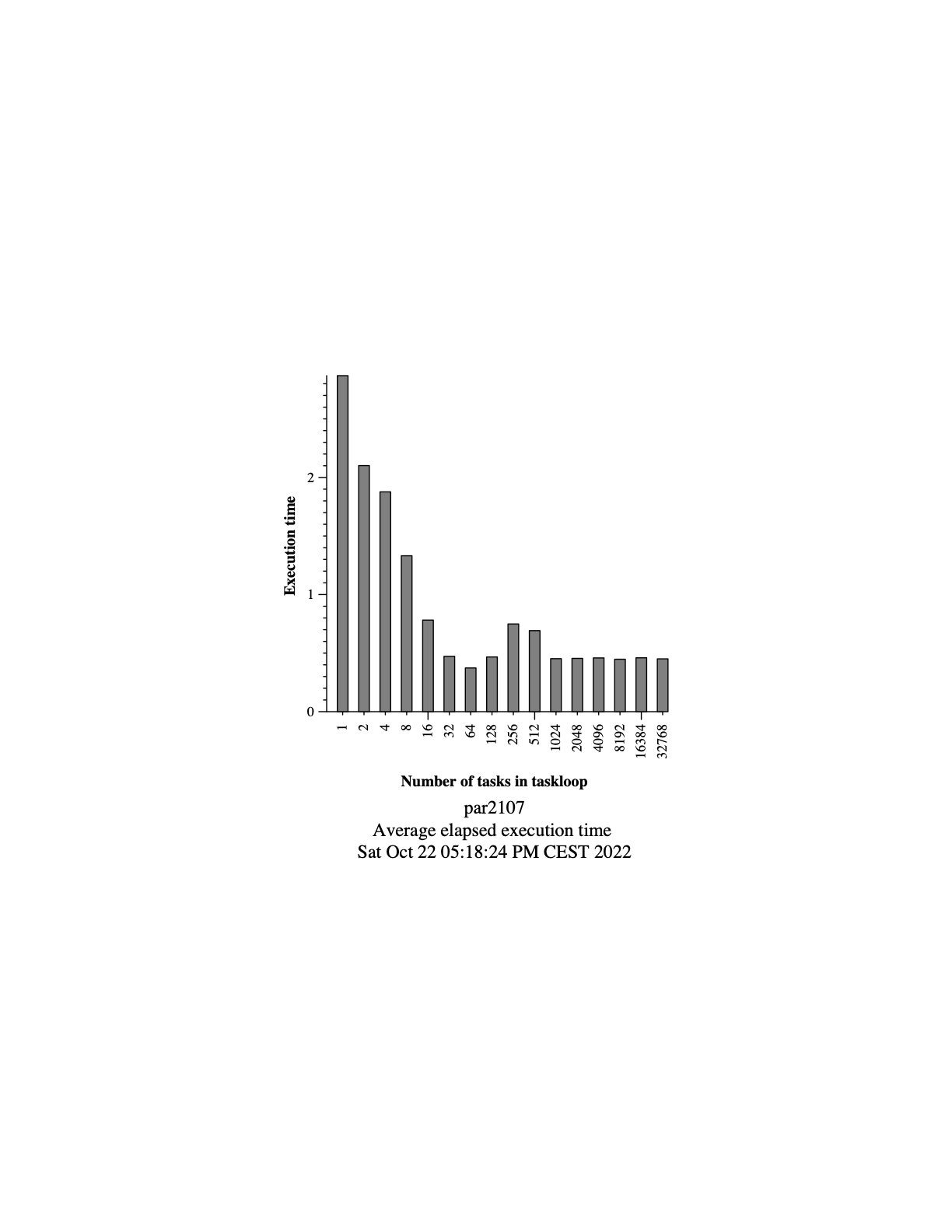
| void mandelbrot(...) {  #pragma omp parallel  #pragma omp single  for (int row = 0; row < height; ++row) {  #pragma omp taskloop num\_tasks(user\_param)   for (int col = 0; col < width; ++col) {   }  }  } |
| --- |

Row strategy

| void mandelbrot(...) {   #pragma omp parallel #pragma omp single   #pragma omp taskloop num\_tasks(user\_param)   for (int row = 0; row < height; ++row) {   for (int col = 0; col < width; ++col) {   }   }  } |
| --- |

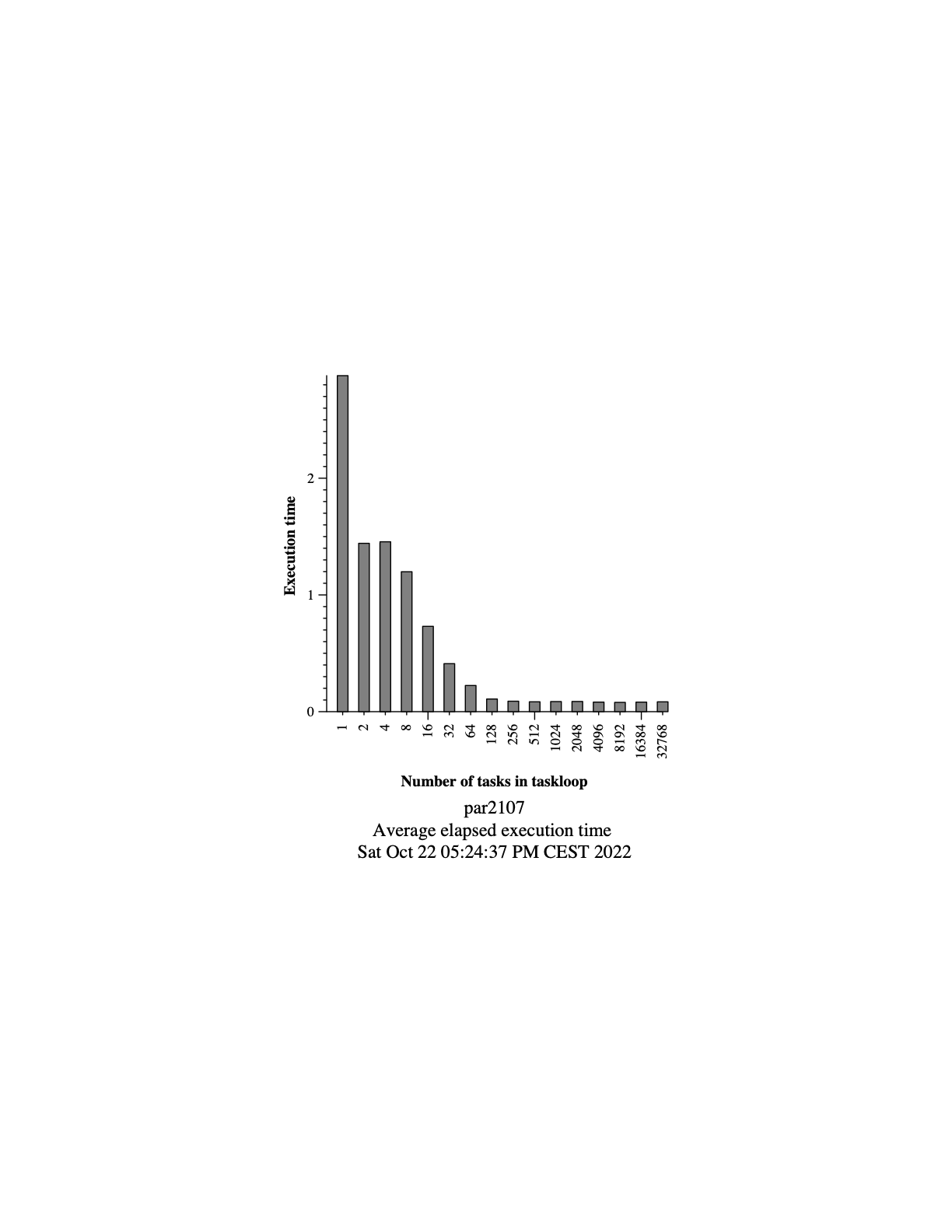
These modifications were made because the script uses the -u option of the program to send the number of tasks as an argument, which is stored in the user\_param variable.

Point strategy graph:



We can see that the execution time descends up until the 64 tasks mark, and then it goes up at 128 till 512 and stabilizes at 1024. This last curve at the end could be due to some kind of overhead of using that many tasks.

Row strategy graph:



For the Row strategy, there was a very rapid descent from 1 to 2 tasks, at about half of the execution time, and it kept descending from there till the 128-256 tasks per taskloop mark.

We can conclude that the optimal granularity for the strategies is between 64-128 tasks per taskloop, using fewer tasks would only increase the execution of the program.

# Conclusions

From the making of this project, we have learnt how different aspects of the parallelization of our code can affect its performance significantly. From the dependencies generated between different tasks, to the number of tasks executed and the synchronization overheads that are generated. From what we have seen we can’t conclude that our version is the best possible parallelization for this piece of code, but we can conclude that the best version that we have done is the row version one as it has the best results.