PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

Group 13-03

Carlos Bárcenas Holguera

Miquel Gómez i Esteve

### Facultat d’Informàtica de Barcelona - UPC

### March 2018

### Spring 2017-2018

#### **Index**

#### 

#### **Part I: Task granularity analysis with Tareador……………..3**

#### **Part II: OpenMP task-based parallelization………………....5**

#### **Part III: OpenMP taskloop-based parallelization…………...7**

#### **Part IV: OpenMP for-based parallelization……………….....9**

#### 

#### 

#### **Part I: Task granularity analysis with Tareador**

We are going to analyze the task granularity of the non-graphical version of our program with tareador. We will compare the two possible task distributions introduced in the deliverable: Row and Point.

These next pictures, have been obtained with tareador and we can see how the tasks are distributed.

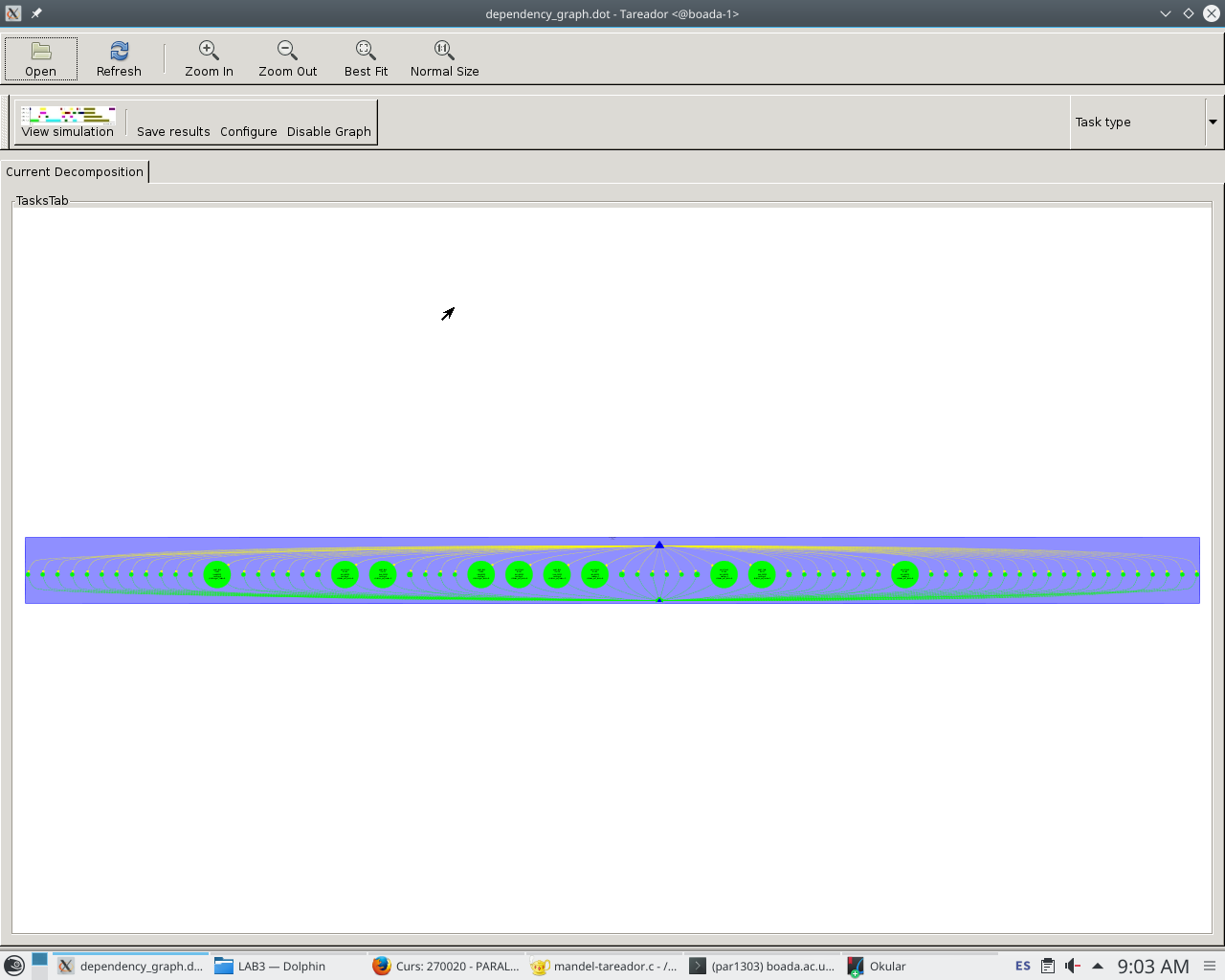


Figure 1: Point task dependency graph

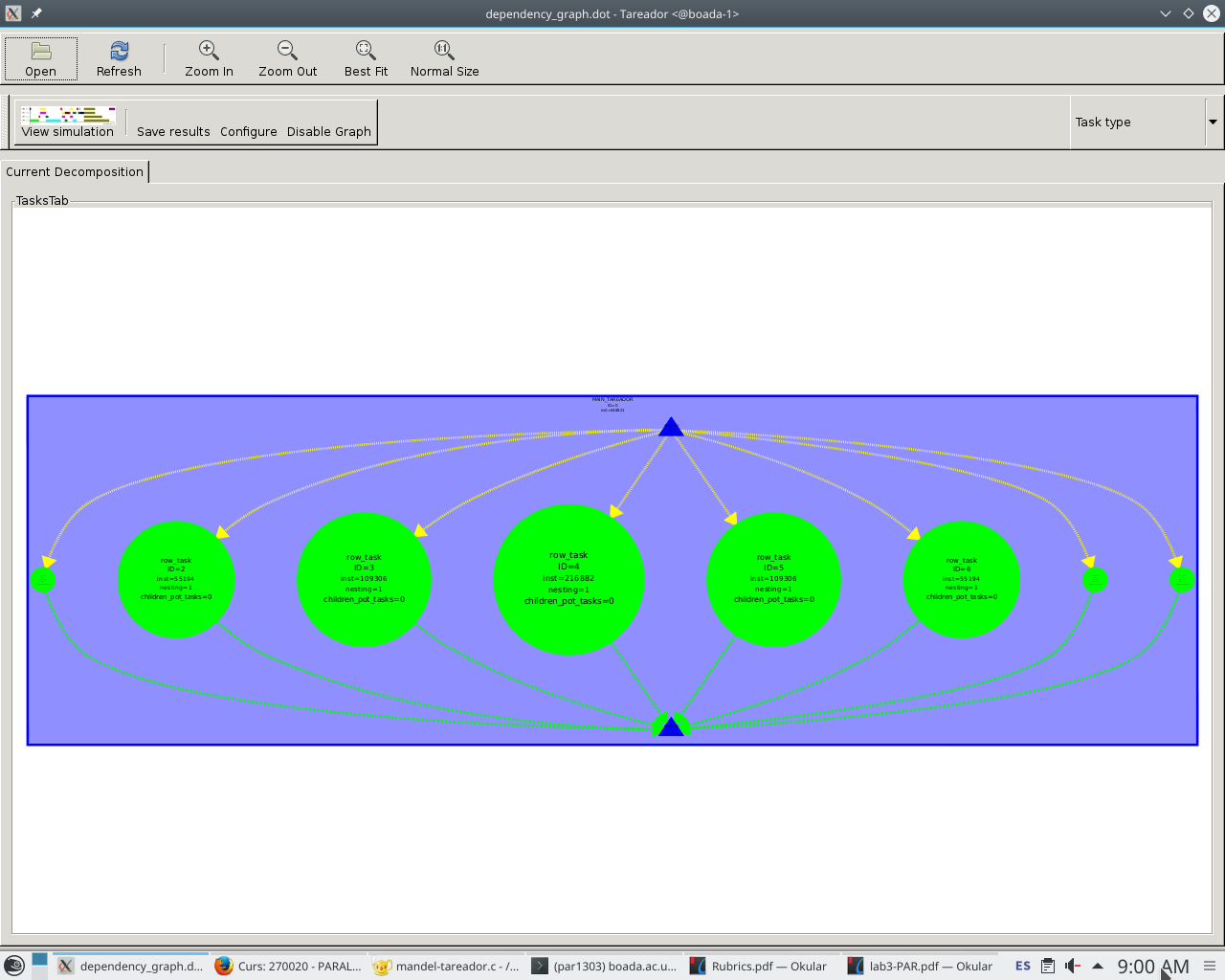
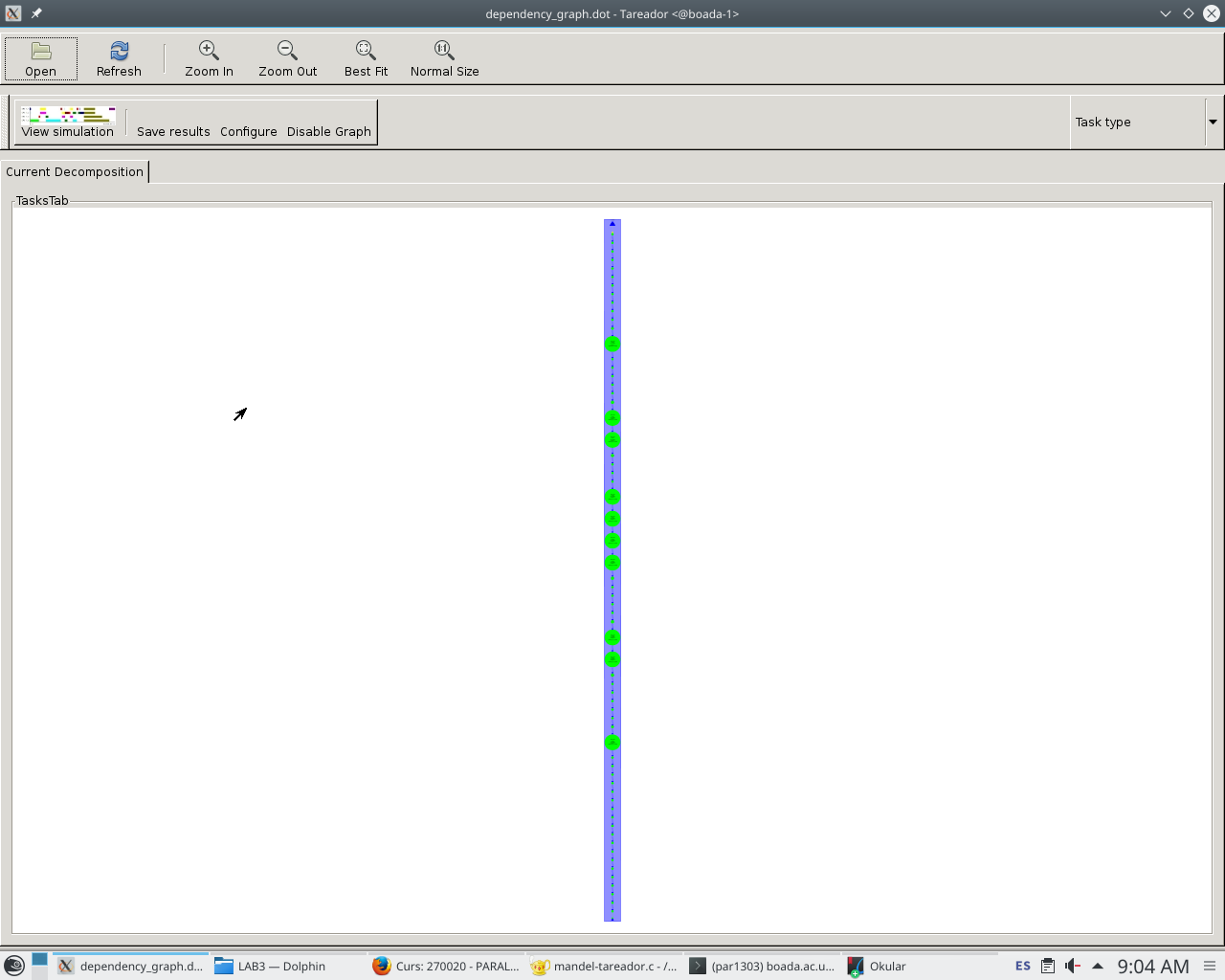


Figure 2: Row task dependency graph

In both cases, the main issue we will have is that there is a huge disbalance between tasks, as some of them require more work than others.

Now we will analyze the graphical version of the program with tareador. And as shown in the following pictures (Figures 3 and 4), there is a clear dependency. This dependencies are caused by a variable called X11\_COLOR\_FAKE.

The code causing the dependency is shown in Figure 5, specifically the functions XSetForeground(...) and XDrawPoint(...). As a solution to protect the confliction zone, we added a #pragma omp critical zone.



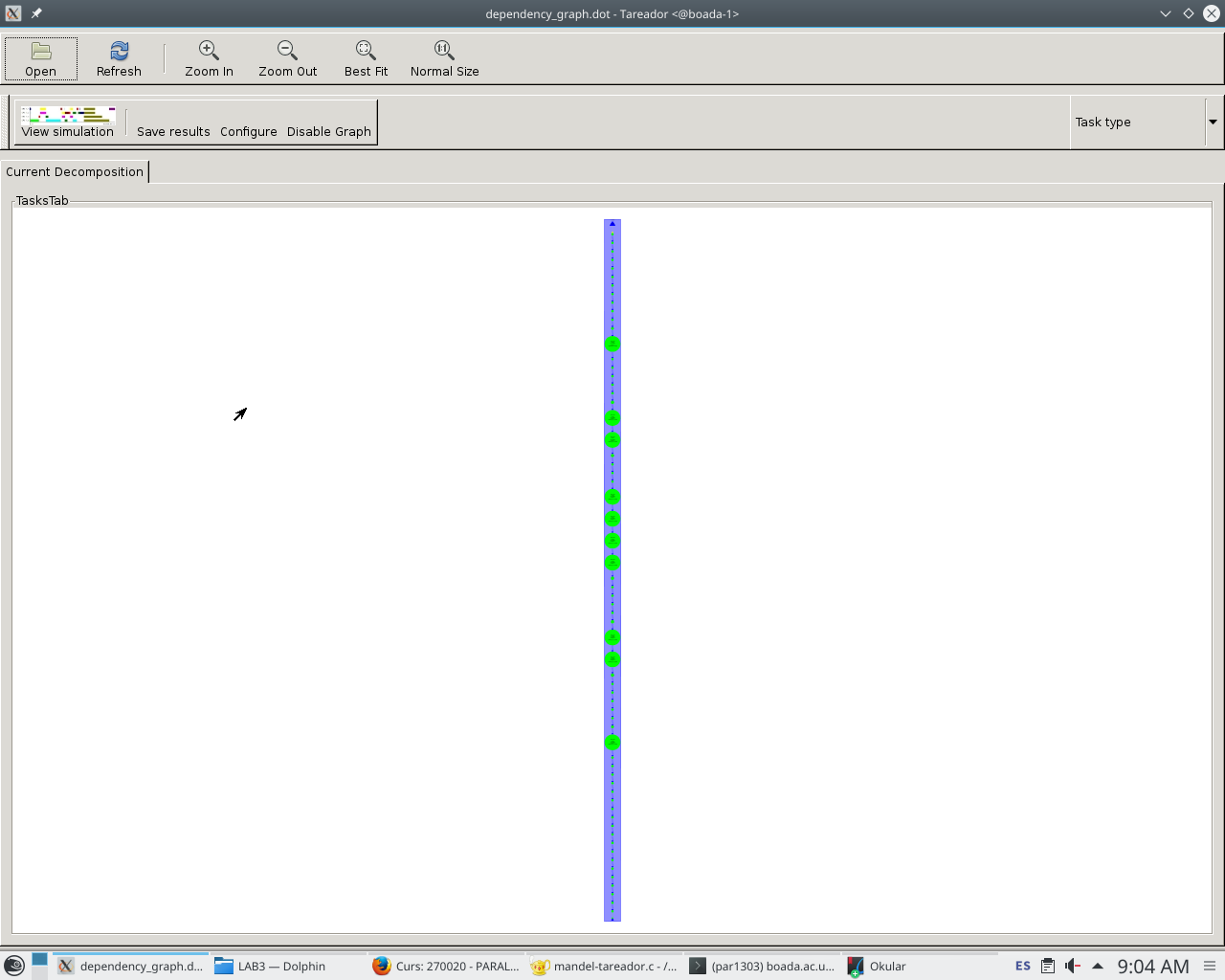
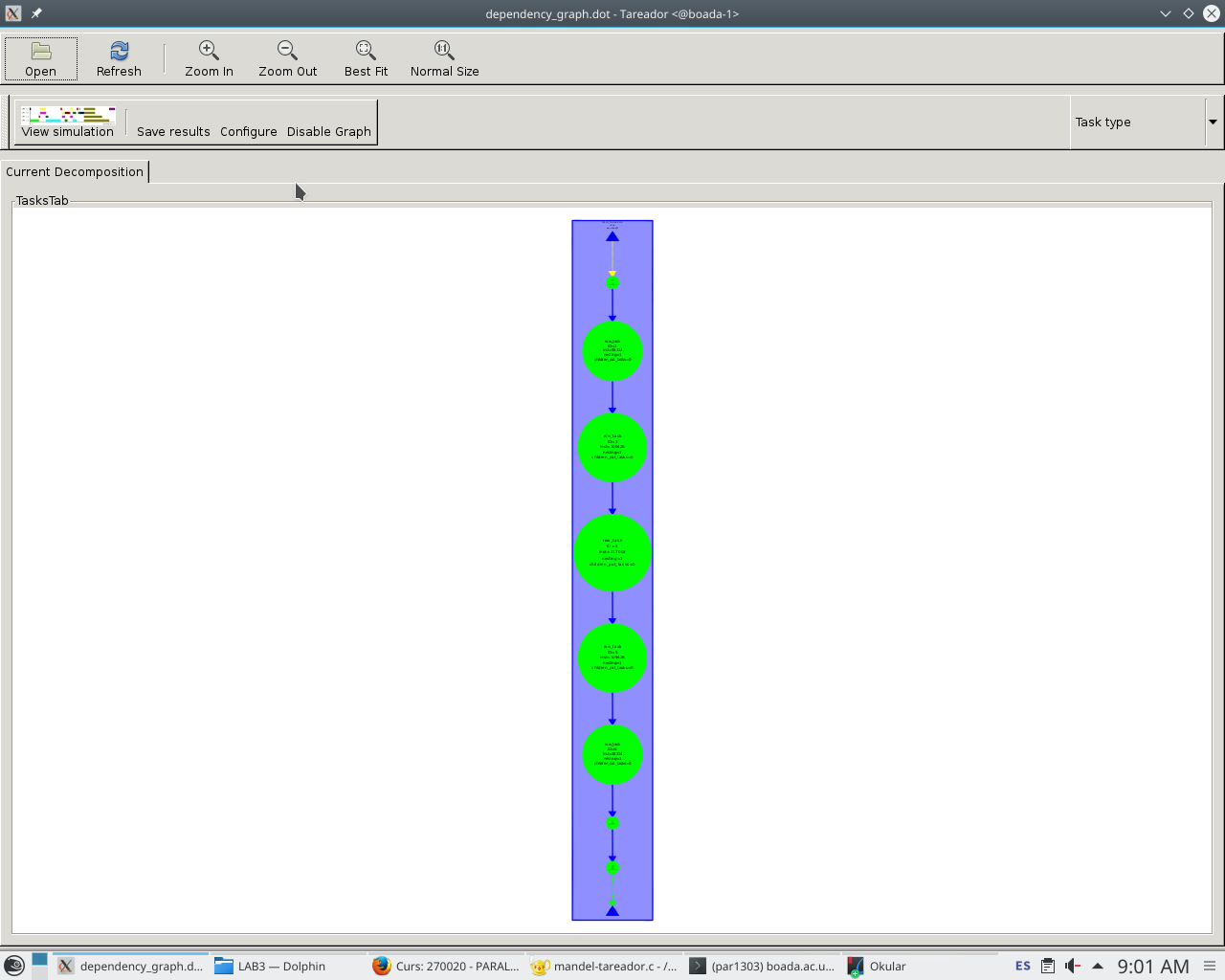


Figure 3:

Row task dependency graph

Figure 4:

Point task Dependency graph

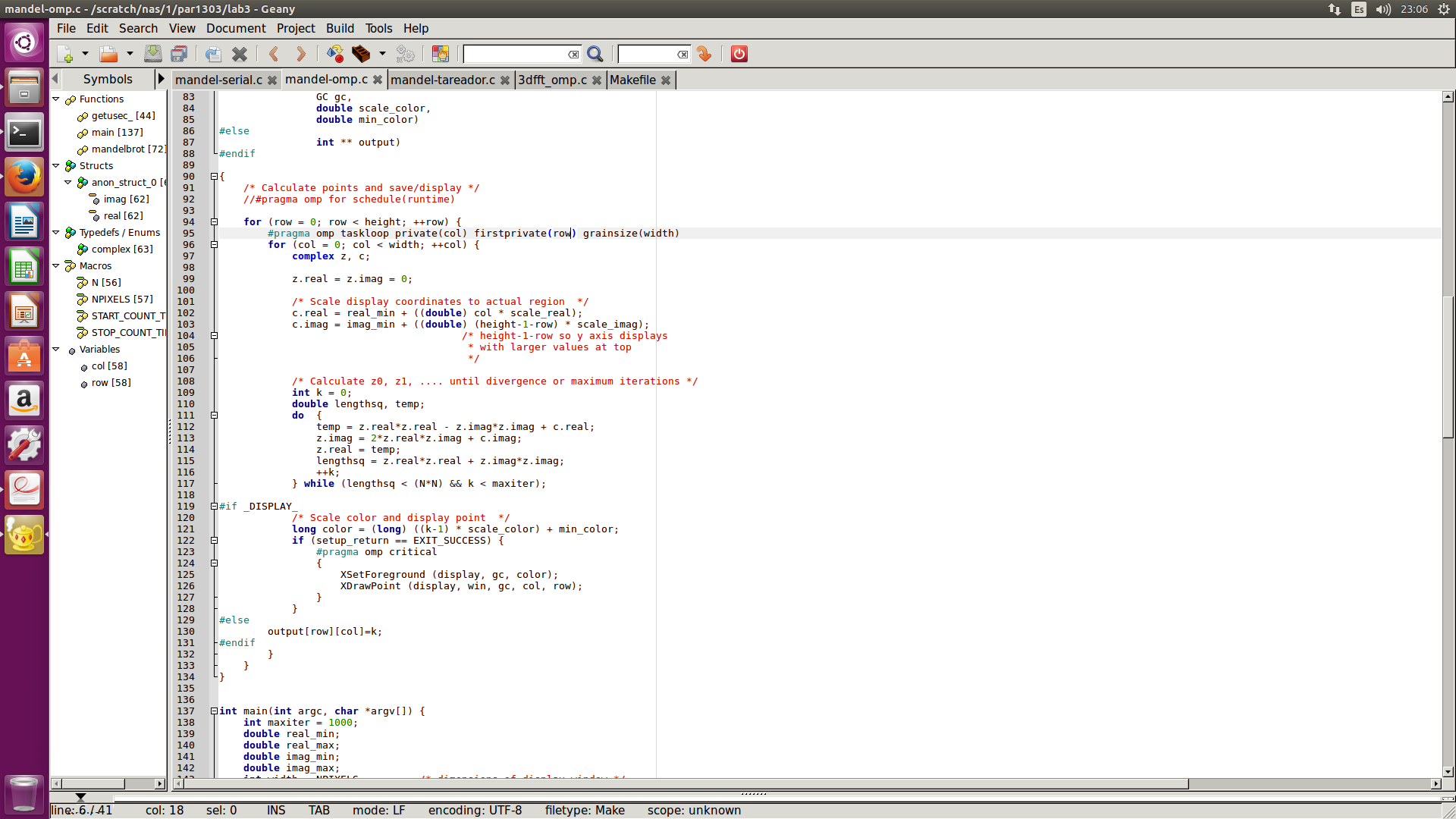


Figure 5: Code causing dependencies in the graphical version

#### 

#### 

#### **Part II: OpenMP task-based parallelization**

We will now parallelize the code using tasks. The first thing to do is to modify the codes as shown in figures 6 (row distribution) and 7 (point distribution). We also needed to make some variables private or firstprivate so that he code would still work properly.

Figure 6: Code for Row Decomposition (Task)

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

Figure 7: Code for Point Decomposition (Task)

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

Now that we have seen the structure of the program, we must now analyze it’s behaviour.

**Row Decomposition:**

When looking at the speed-up and time plots from figure 8, we can clearly see that the more threads we add, the program reduces it’s execution time and increases it’s speed-up in a nearly linear way. We can see that the row distribution would be a good way to parallelize this program since we do not create many tasks that could force us to have much more overhead.

**Point Decomposition:**

Observing the plots in figure 9, it is clear that the point distribution is much worse than the row distribution. We an see a slight improvement when we have 4 threads, but if we keep adding more of them, the program with this distribution becomes very inefficient. One of the big issues that don’t allow us to reduce the execution time would be the large overhead caused by the creation of a lot of tasks.

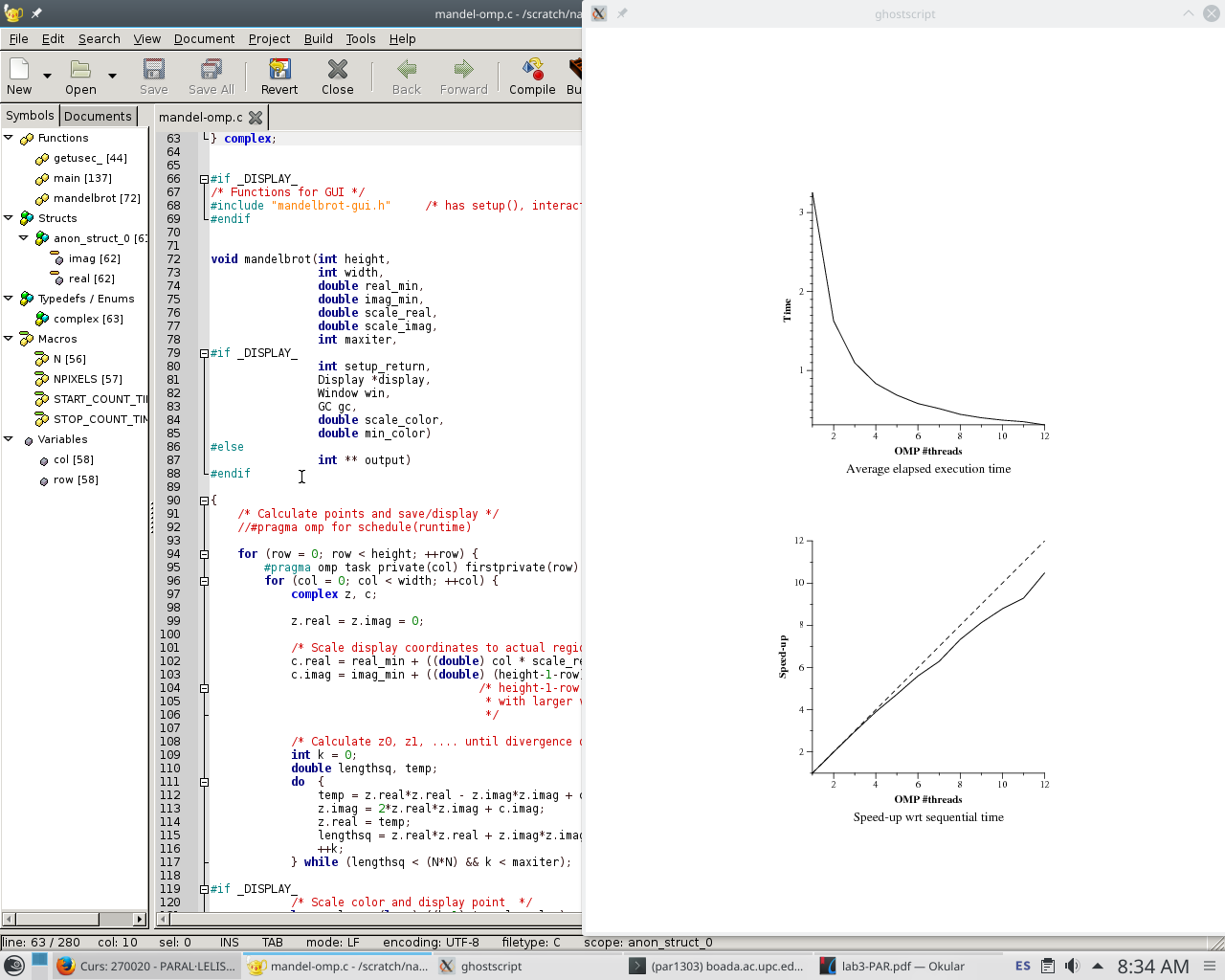


Figure 8:

Speedup and execution time plots for row distribution.

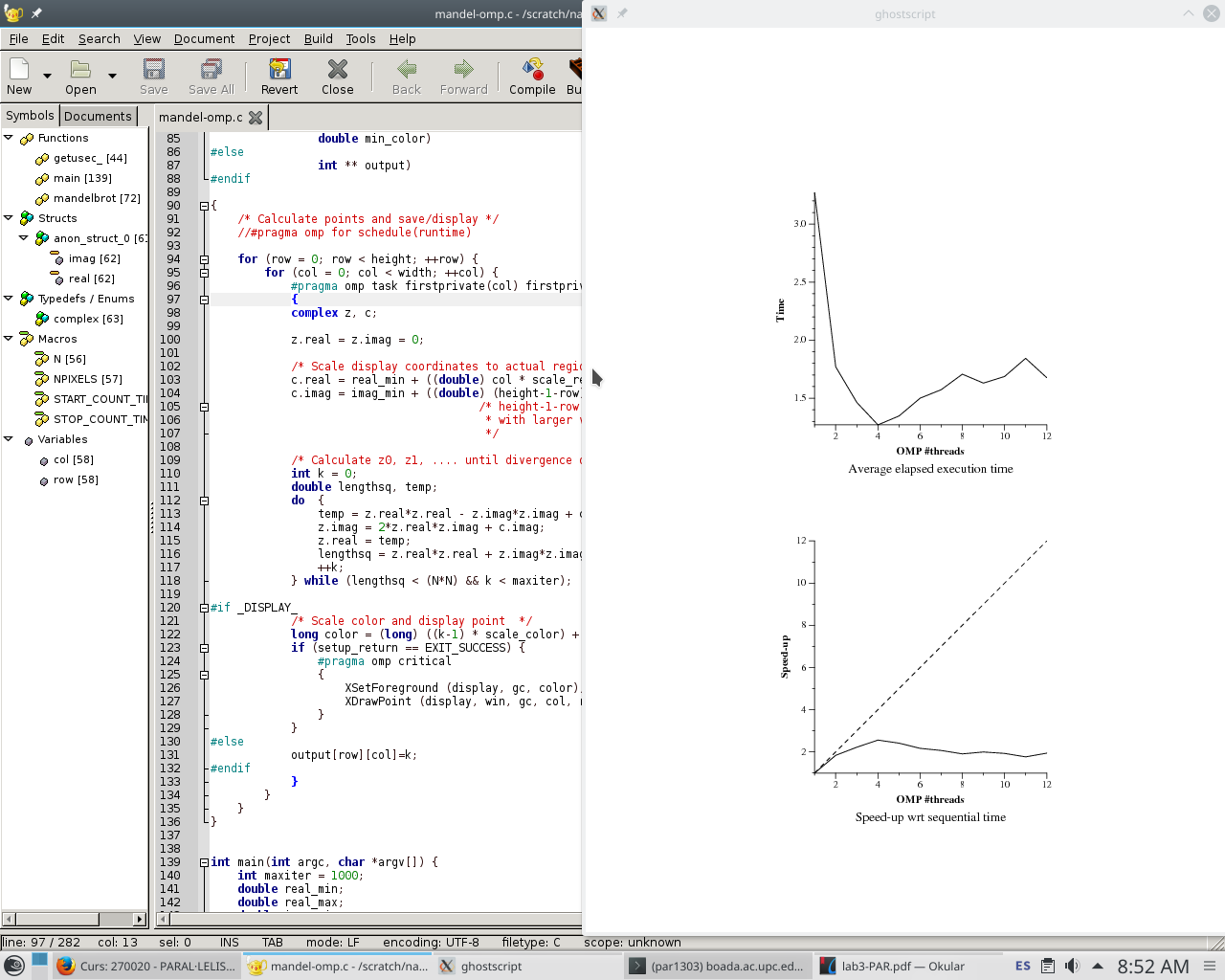


Figure 9:

Speedup and execution time plots for point distribution.

When checking each of the alternatives strong scalability, the row distribution is very scalable while the point distribution is not.

#### **Part III: OpenMP taskloop-based parallelization**

We will now use a different strategy to parallelize the code, we will use the taskloop construct. First of all we have to parallelize the code with the two strategies. In figure 10 we can see the code for the row distribution and in figure 11 for the point distribution.

After some tests we decided that the best option would be to use the grainsize(20) and grainsize(5) construct in the taskloop declaration for the point and row decomposition respectively.

Figure 10: Code for Row Decomposition (Taskloop)

|  |
| --- |
| #pragma omp taskloop grainsize(5)  for (row = 0; row < height; ++row) {  for (col = 0; col < width; ++col) {....}  } |

Figure 11: Code for Point Decomposition (Taskloop)

|  |
| --- |
| for (row = 0; row < height; ++row) {  #pragma omp taskloop grainsize(20)  for (col = 0; col < width; ++col) {....}  } |

It’s time to analyze the two decompositions with taskloop. The plots below (figures 12 and 13) have been obtained when executing the program with the two possible decompositions. We can see that in both cases when analyzing the strong scalability, both of them are scalable. The row distribution seems to be a bit more scalable since it has a higher speed-up in most cases, however, this same distributions also seems to be unstable since the values have varied with the last executions, the ones that use more threads.

We thought that the reason behind the point distribution being less scalable must be again the overhead generated when we create that many tasks.

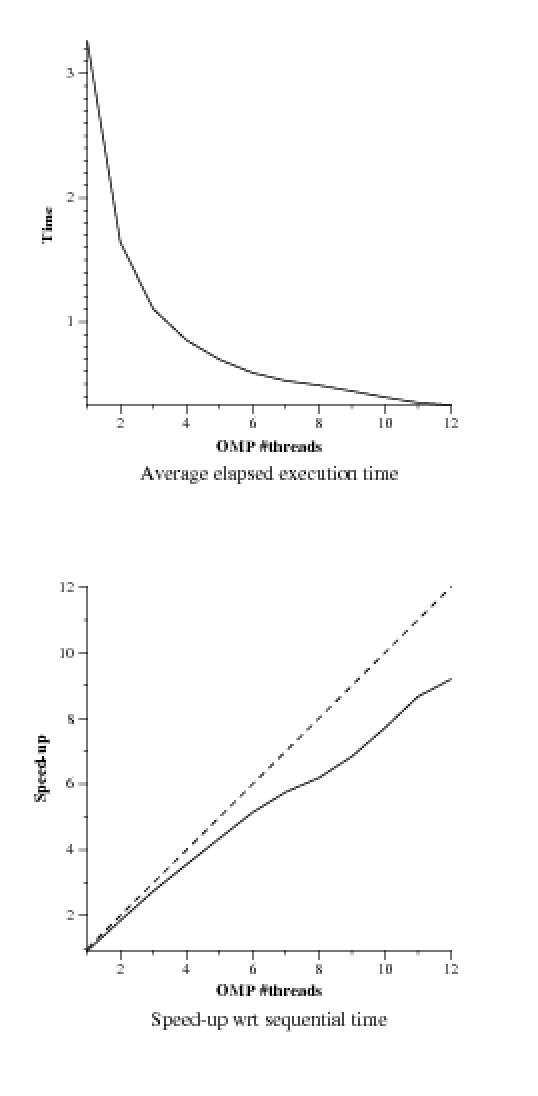


Figure 12:

Speedup and execution time plots for point distribution.

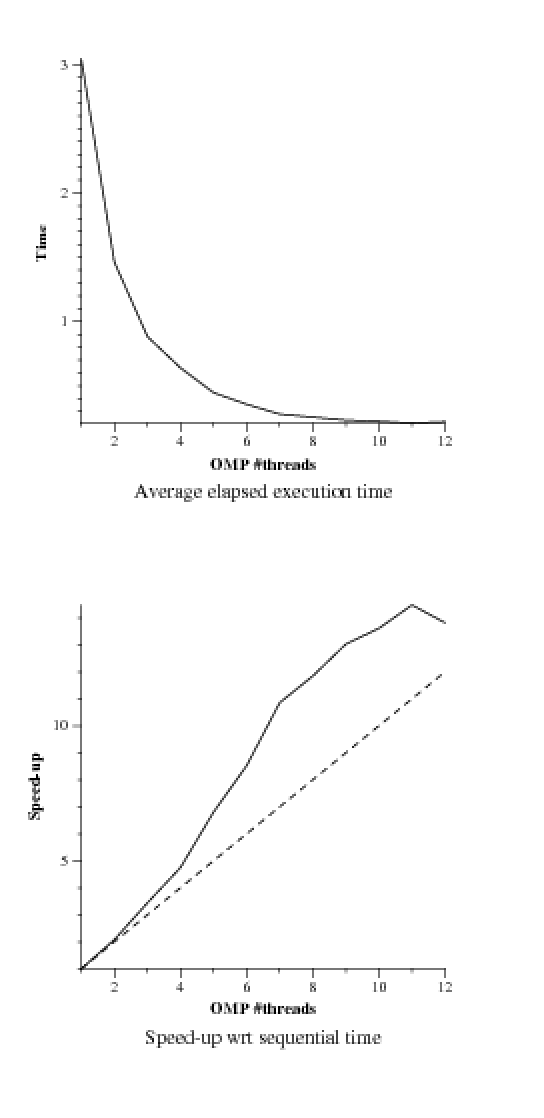


Figure 13:

Speedup and execution time plots for row distribution.

#### **Part IV: OpenMP for-based parallelization**

We will use a different parallelization strategy now, now we use a for strategy with different schedule strategies or policies. In figure 14, we can see the changes in the code, to row decomposition strategy. In figure 15, we can see the cagnes in the code, to point decomposition.

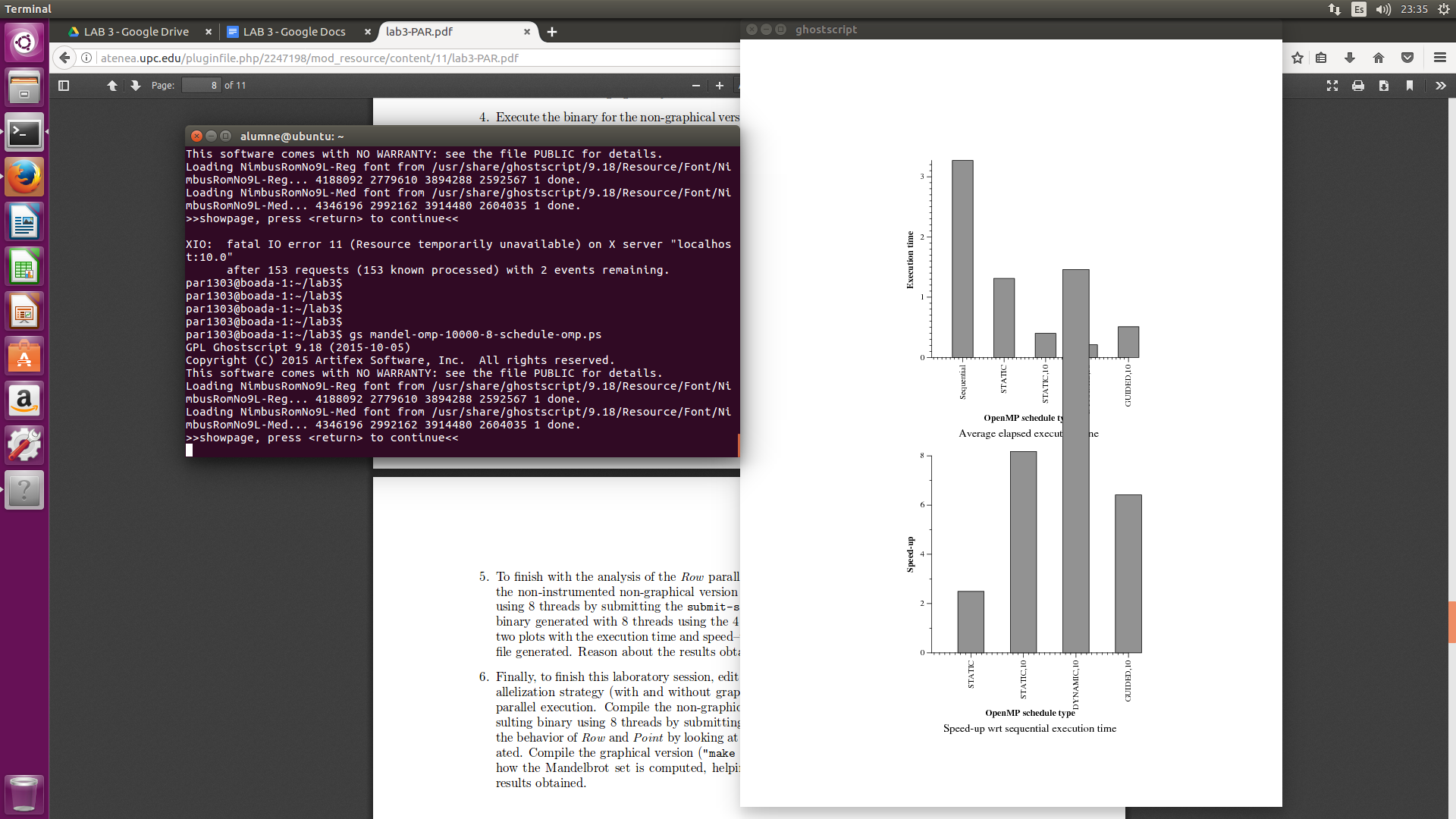
Figure 14: Code for Row Decomposition (For)

|  |
| --- |
| #pragma omp for schedule(runtime)  for (row = 0; row < height; ++row) {  for (col = 0; col < width; ++col) {....}  } |

Figure 15: Code for Point Decomposition (For)

|  |
| --- |
| for (row = 0; row < height; ++row) {  #pragma omp for schedule(runtime)  for (col = 0; col < width; ++col) {....}  } |

**Row Decompositon:**

Figure 16: Execution time and speed-up Row version

Conclusions of Row Decomposition (For):

- STATIC: As we can see in the graphics the static version divides the number of loops to execute in the number of threads which in this case is 8, and the improvement is good but is not the most significant in comparison with the rest of options because each thread executes about 100 iterations.

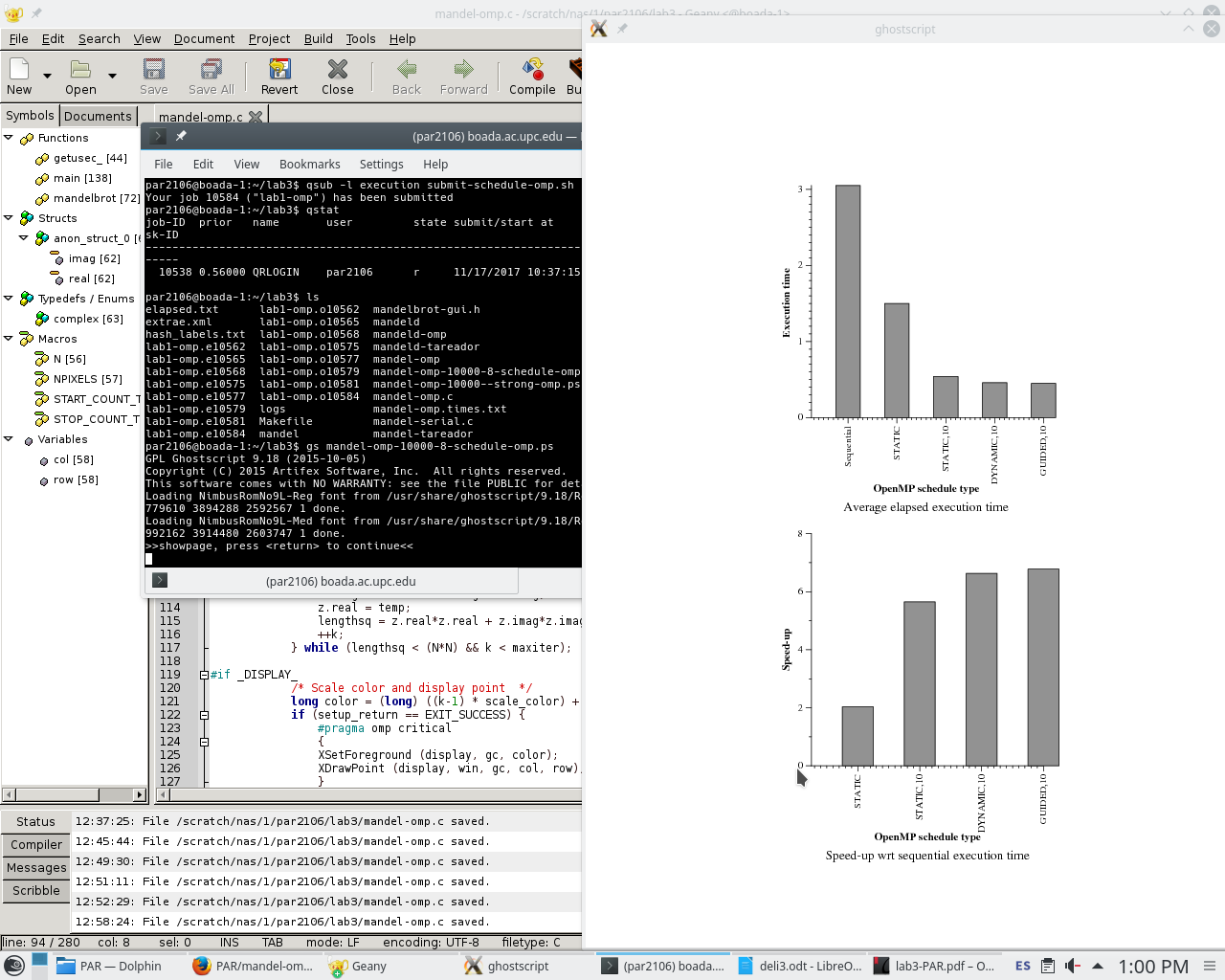
- STATIC, 10: The improvement is better than in the STATIC version, in this case each thread executes 80 iterations, dividing the work and scheduling it in runtime, which is better and makes all the execution more fast.

- DYNAMIC, 10: The chunks have been divided in runtime among the threads in chunks of size 10, and taking advantage of the as soon as a thread is ready can immediately execute the task, and that’s why the speed-up is so impressive.

- GUIDED, 10: In this case the chunk size decreases as the threads grab iterations, and all of them are at least of size 10, balancing all the work, but in this case the option is better than the static one, but not better than the static,10 or dynamic,10.

In this case the two best options are the Static,10 and Dynamic,10 because of the capacity of dividing small chunks among the ready threads, and also because the size of the chunks is the optimal one, avoiding a big overhead and solving the imbalance problems.

**Point Decomposition:**

****Figure 17: Execution time and speed-up Point version.

Conclusions of Point Decomposition (For):

As we saw in the individual conclusions in the Row decomposition, we know how each method (static / static,10 / dynamic, 10 / guided,10) works and in this case the goal is the same but we don’t see a big difference due to the Point decomposition and the big number of tasks generated. The best results are coming from the Dynamic,10 and Static,10 due the efficiency of dividing the work among the threads and the correct number of chunks. The guided option don’t show a big difference because the overhead added due to resizing the chunks in running-time.

Finally we make a table with information extracted from the individual executions of the schedules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Static** | **Static, 10** | **Dynamic, 10** | **Guided, 10** |
| **Running average time per thread** | 413,546 ns | 450,435 ns | 441,028 ns | 420,786 ns |
| **Execution unbalance (average**  **time divided by maximum time)** | 0.31 ns | 0.86 ns | 0.87 ns | 0.46 ns |
| **SchedForkJoin (average time per**  **thread or time if only one does)** | 1,399,25 ns | 456,1 ns | 371,34 ns | 815,534 ns |

As we can see in the chart, the average time per thread is better the Static schedule, but not

very different from the guided. About the execution unbalance, the bigger means better, and as we can see the dynamic and static,10 are the best results.