PAR Laboratory Assignment

Lab 5: Geometric (data) decomposition: solving the heat equation

Group 13-03

Carlos Bárcenas Holguera

Miquel Gómez i Esteve

### Facultat d’Informàtica de Barcelona - UPC

### March 2018

### Spring 2017-2018

#### **Index**

#### 

**Part I:**

**Analysis with Tareador......................................................3**

**Part II:**

**Parallelization of Jacobi with OpenMP…………………...11**

**Part III:**

**Parallelization of Gauss-Seidel with OpenMP……...........15**

#### 

**Part I: Analysis with tareador**

We will try to analyse the program with tareador. First we make an execution by default to see the big tasks. We start with the **Jacobi** version:

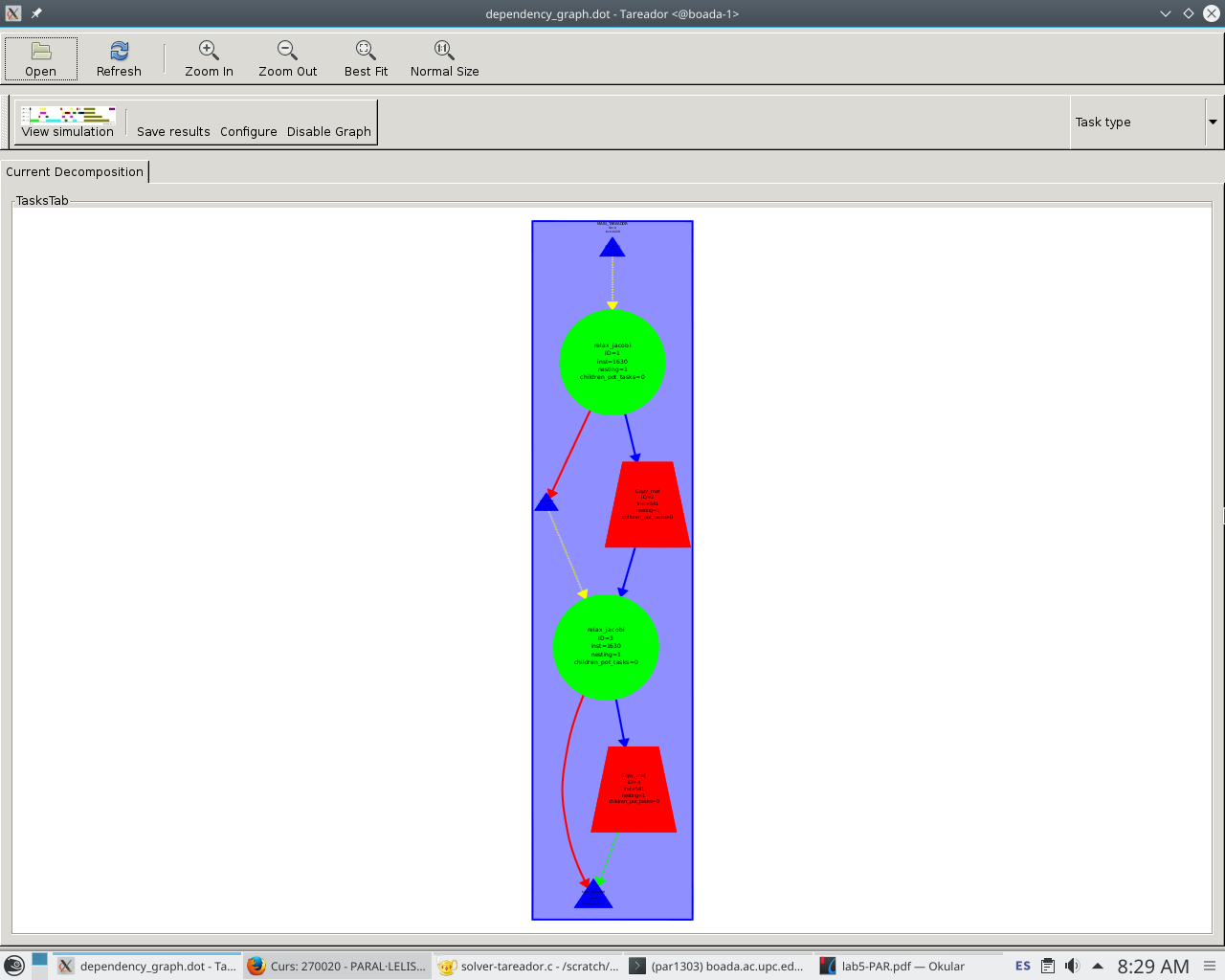


Figure 1: Jacobi tareador execution without tasks.

We can appreciate two principle tasks, relax\_jacobi and copy\_mat.

First we must add the definitions for the inner tasks to see how tasks are distributed. We define the “tareador\_start\_task("inner\_jacobi");” & “tareador\_end\_task("inner\_jacobi");” between the inner for. The result is this:

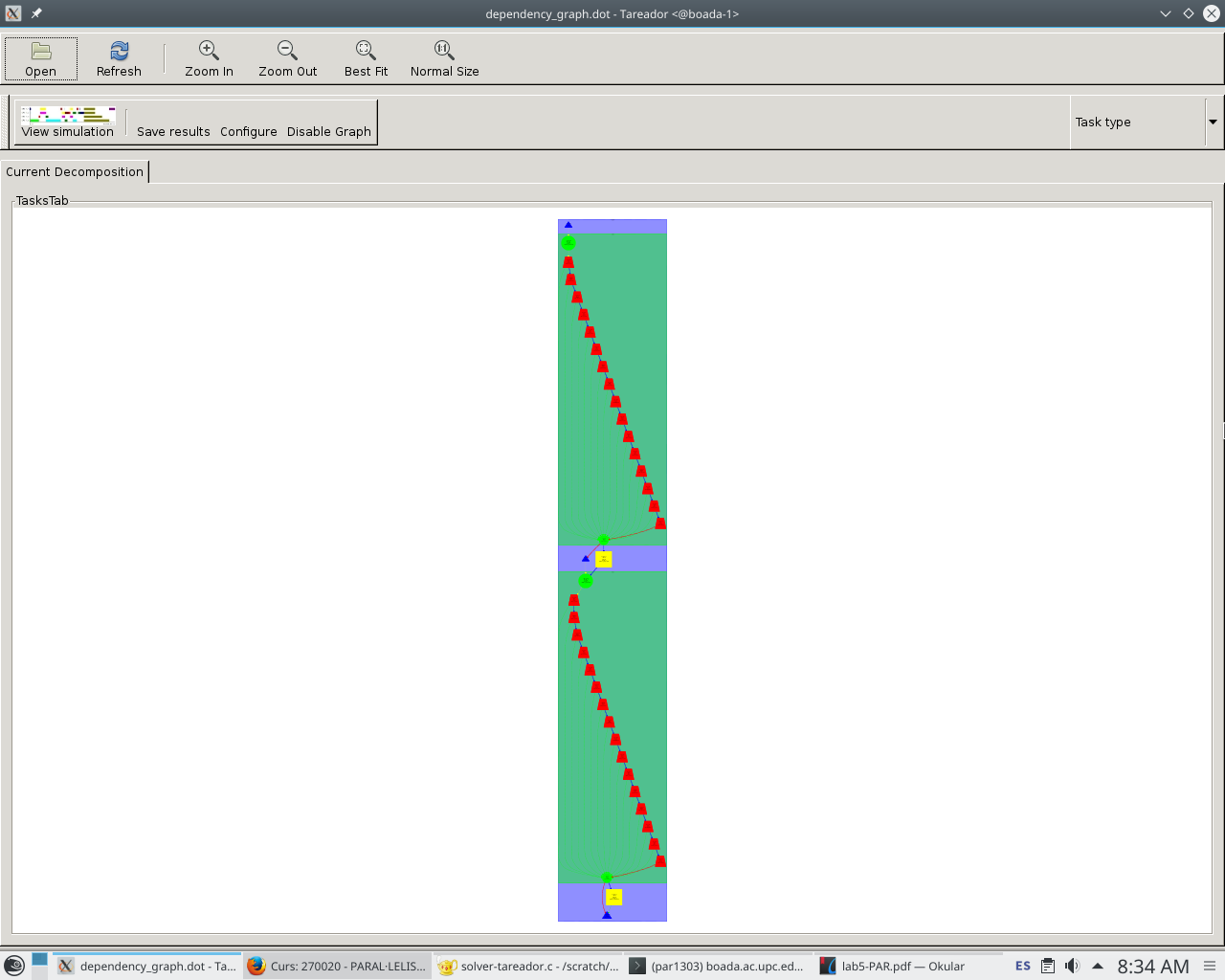


Figure 2: Jacobi with inner tasks defined (Tareador)

We can appreciate that there’s a big dependency, that dependency is the variable sum. To find out if there’s the only dependency we use disable, to disable the object.

The result after the application of the disable is this:

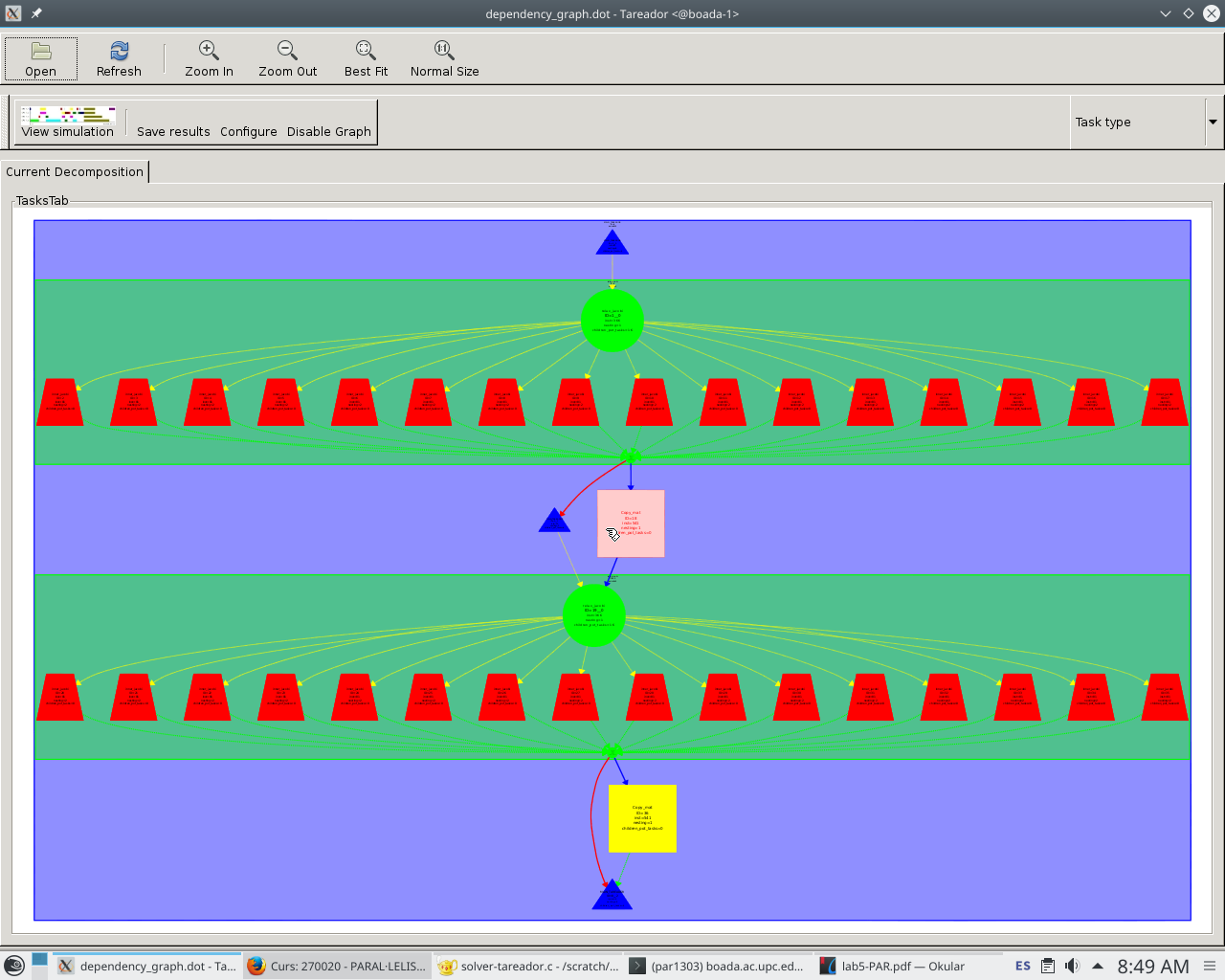


Figure 3: Jacobi with disable\_object(&sum) (Tareador)

That’s the code with the application of the disable:

|  |
| --- |
| double relax\_jacobi (double \*u, double \*utmp, unsigned sizex, unsigned sizey)  {  double diff, sum=0.0;    int howmany=1;  for (int blockid = 0; blockid < howmany; ++blockid) {  int i\_start = lowerb(blockid, howmany, sizex);  int i\_end = upperb(blockid, howmany, sizex);  for (int i=max(1, i\_start); i<= min(sizex-2, i\_end); i++) {  for (int j=1; j<= sizey-2; j++) {  tareador\_start\_task("inner\_jacobi");  utmp[i\*sizey+j]= 0.25 \* ( u[ i\*sizey + (j-1) ]+ // left  u[ i\*sizey + (j+1) ]+ // right  u[ (i-1)\*sizey + j ]+ // top  u[ (i+1)\*sizey + j ]); // bottom  diff = utmp[i\*sizey+j] - u[i\*sizey + j];  tareador\_disable\_object(&sum);  sum += diff \* diff;  tareador\_enable\_object(&sum);  tareador\_end\_task("inner\_jacobi");  }  }  }  return sum;  } |

Figure 4: Jacobi with inner tasks defined and sum disable.

Now we analyze the same as before but with **Gauss-Seidel** code:

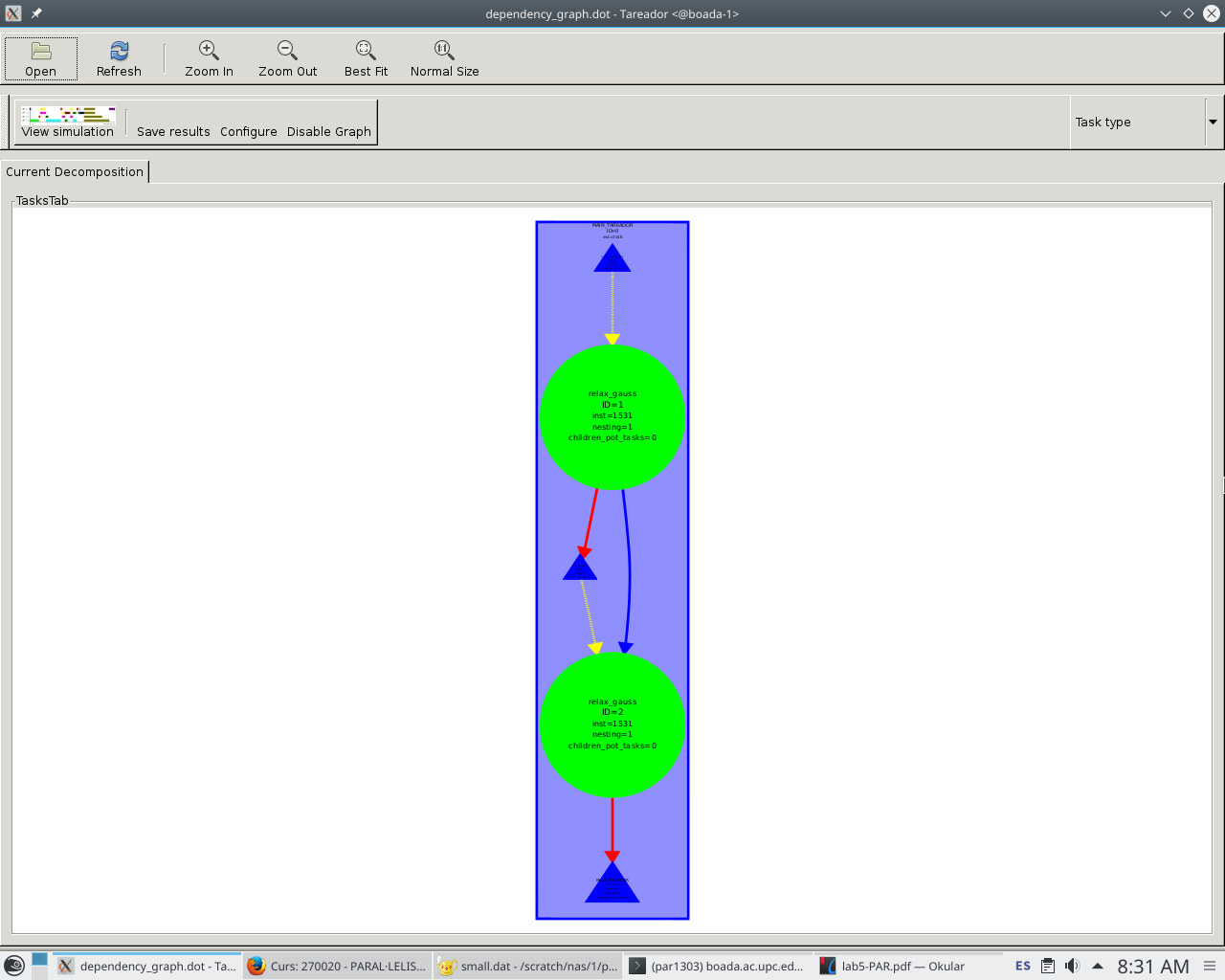


Figure 4: Gauss-Seidel tareador execution without tasks.

We can appreciate one principle task, relax\_gauss.

First we must add the definitions for the inner tasks to see how tasks are distributed. We define the “tareador\_start\_task("inner\_gauss");” & “tareador\_end\_task("inner\_gauss");” between the inner for. The result is this:

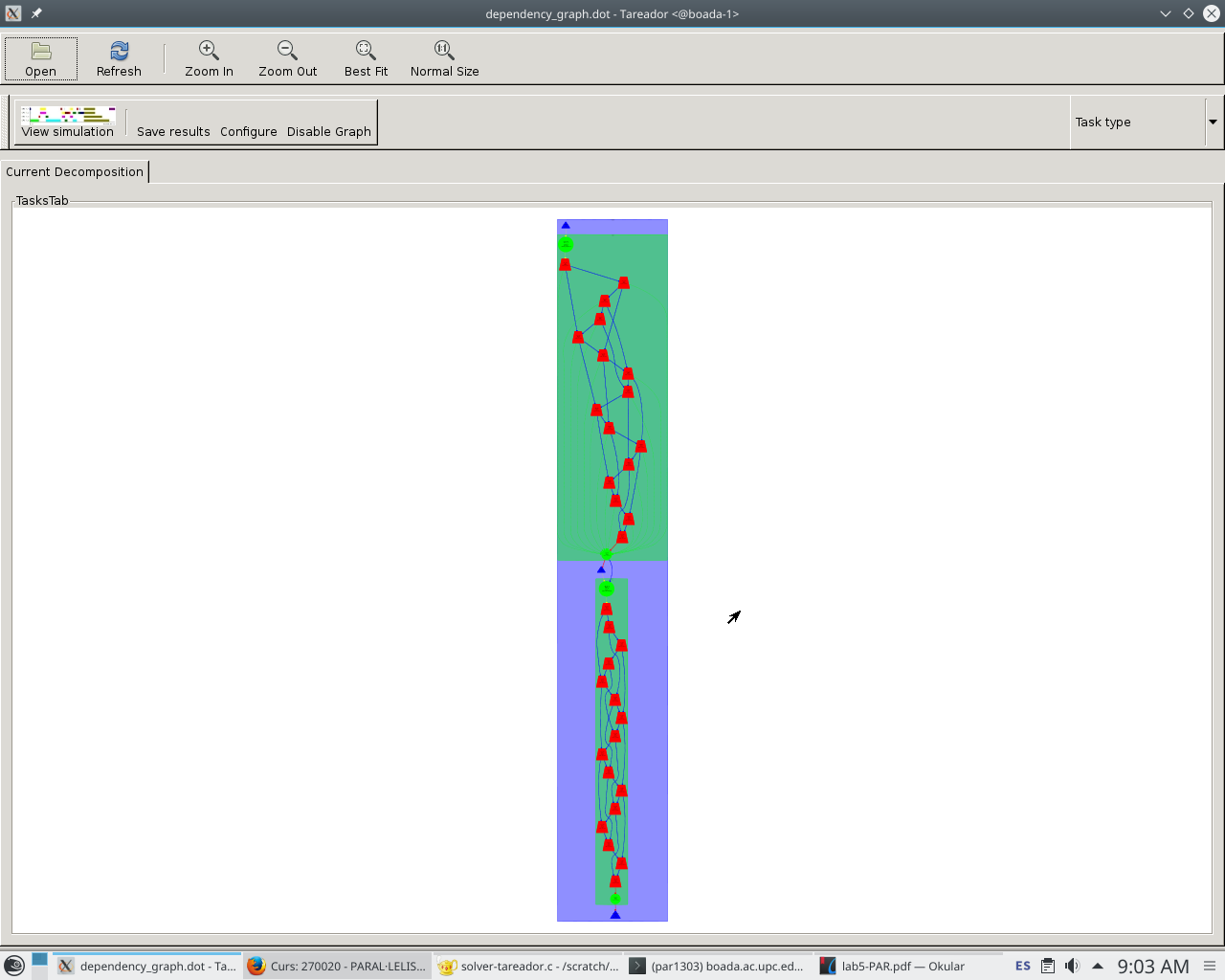


Figure 5: Gauss-Seidel with inner tasks defined (Tareador)

We can appreciate that there’s a big dependency, that dependency is the variable sum. To find out if there’s the only dependency we use disable, to disable the object.

The result after the application of the disable is this:

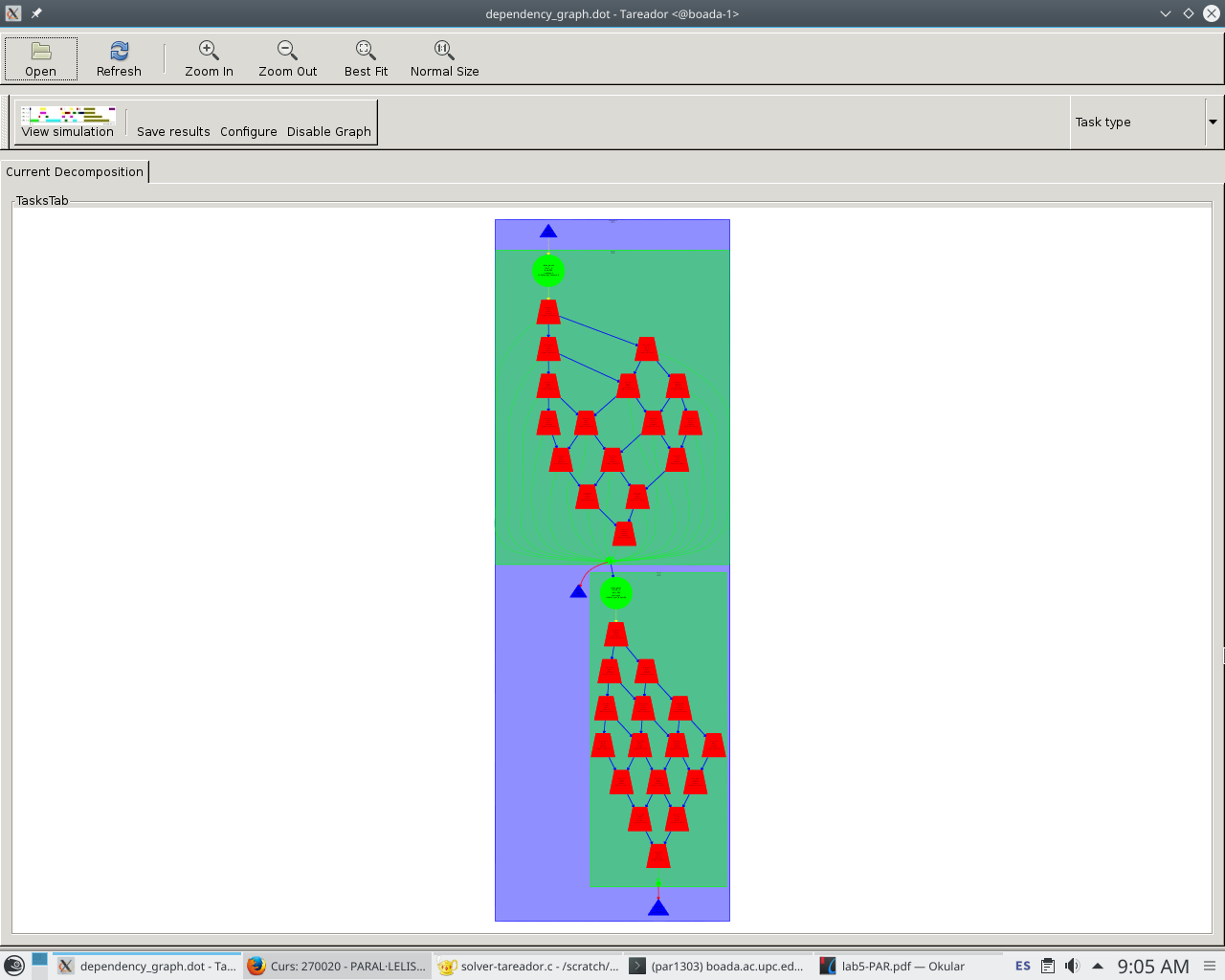


Figure 6: Gauss with disable\_object(&sum) (Tareador)

That’s the code with the application of the disable:

|  |
| --- |
| double relax\_gauss (double \*u, unsigned sizex, unsigned sizey)  {  double unew, diff, sum=0.0;  int howmany=1;  for (int blockid = 0; blockid < howmany; ++blockid) {  int i\_start = lowerb(blockid, howmany, sizex);  int i\_end = upperb(blockid, howmany, sizex);  for (int i=max(1, i\_start); i<= min(sizex-2, i\_end); i++) {  for (int j=1; j<= sizey-2; j++) {  tareador\_start\_task("inner\_gauss");  unew= 0.25 \* ( u[ i\*sizey + (j-1) ]+ // left  u[ i\*sizey + (j+1) ]+ // right  u[ (i-1)\*sizey + j ]+ // top  u[ (i+1)\*sizey + j ]); // bottom  diff = unew - u[i\*sizey+ j];  tareador\_disable\_object(&sum);  sum += diff \* diff;  tareador\_enable\_object(&sum);  u[i\*sizey+j]=unew;  tareador\_end\_task("inner\_gauss");  }  }  }  return sum;  } |

Figure 7: Gauss-Seidel with inner tasks defined and sum disable.

In both Jacobi and Gauss-Seidel, we can protect the variable sum from an OpenMp parallelization using the clause reduction in order to guarantee each thread has a private access to variable sum.

We can point out that the Jacobi implementation is more parallelizable than the Gauss-Seidel one.

The times obtained simulating with tareador are the following:

Jacobi:

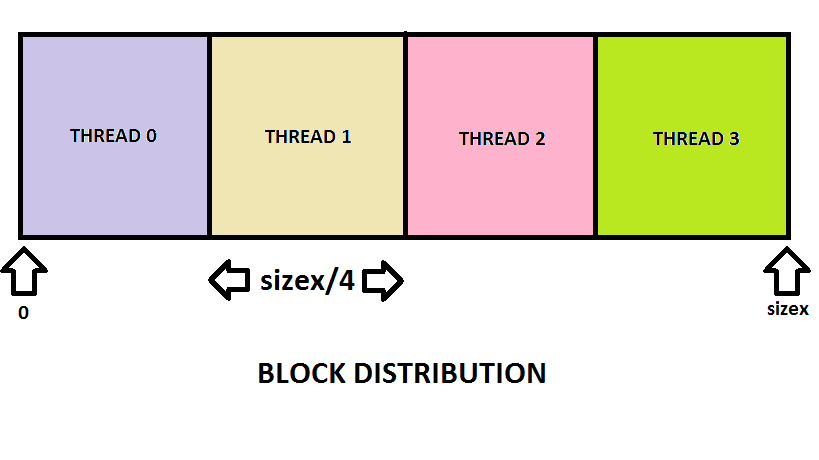
|  |  |
| --- | --- |
| # threads | Temps (microseconds) |
| 1 | 4.492 |
| 2 | 2.920 |
| 3 | 2.434 |
| 4 | 2.150 |
| 6 | 1.948 |
| 8 | 1.948 |

Gauss:

|  |  |
| --- | --- |
| # threads | Temps (microseconds) |
| 1 | 3.212 |
| 2 | 1.862 |
| 3 | 1.480 |
| 4 | 1.412 |
| 6 | 1.330 |
| 8 | 1.330 |

**Part II: Parallelization of Jacobi with OpenMP**

Now we want to parallelize the Jacobi code, to do so first we need to understand the code and how we can optimize it better. First we want to know what kind of data decomposition is in blocks.



The data decomposition strategy followed in this case divides N (the size) between all the threads (4) leaving N/4 for each thread to compute.

As we know by the analysis with tareador we know that we must protect the variable sum with a reduction, and also we know that the variable diff must be protected. So the code is like this:

|  |
| --- |
| double relax\_jacobi (double \*u, double \*utmp, unsigned sizex, unsigned sizey)  {  double diff, sum=0.0;    int howmany=omp\_get\_max\_threads();  #pragma omp parallel for reduction(+:sum) private(diff)  for (int blockid = 0; blockid < howmany; ++blockid) {  int i\_start = lowerb(blockid, howmany, sizex);  int i\_end = upperb(blockid, howmany, sizex);  for (int i=max(1, i\_start); i<= min(sizex-2, i\_end); i++) {  for (int j=1; j<= sizey-2; j++) {  utmp[i\*sizey+j]= 0.25 \* ( u[ i\*sizey + (j-1) ]+ // left  u[ i\*sizey + (j+1) ]+ // right  u[ (i-1)\*sizey + j ]+ // top  u[ (i+1)\*sizey + j ]); // bottom  diff = utmp[i\*sizey+j] - u[i\*sizey + j];  sum += diff \* diff;  }  }  }  return sum;  } |

Figure 8: Jacobi code with #pragma omp parallel

Also we have changed the value for *howmany* because the old value was 4. That value didn’t benefit for the 8 or more threads value, so the execution time was increased for that reason. So we changed the value for *omp\_get\_max\_threads()* so we can benefit for the maximum values of threads.

Also, we change other part of the code but in another function which affects with external behaviour. We parallelized the copy\_mat function, like this:

|  |
| --- |
| void copy\_mat (double \*u, double \*v, unsigned sizex, unsigned sizey)  {  #pragma omp parallel for  for (int i=1; i<=sizex-2; i++)  for (int j=1; j<=sizey-2; j++)  v[ i\*sizey+j ] = u[ i\*sizey+j ];  } |

Figure 9: copy\_mat code with #pragma omp parallel

Now we can analyze the results with the changes we have made. First, we can start with the execution time with 8 threads:

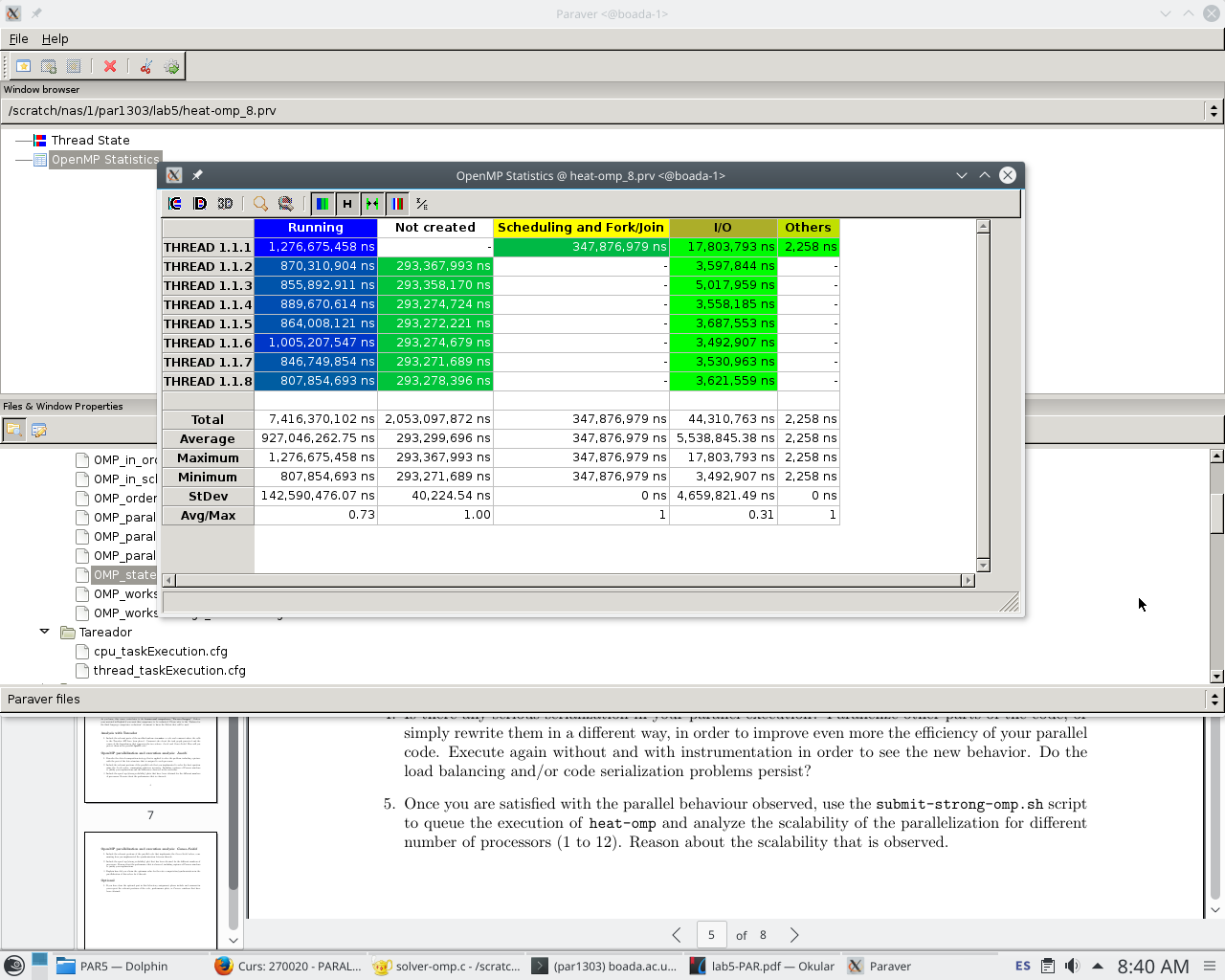


Figure 10: Jacobi execution time

Also we can analyze the trace part with 8 threads, which shows us that there's some balancing problems and all threads don’t work the same amount of time.

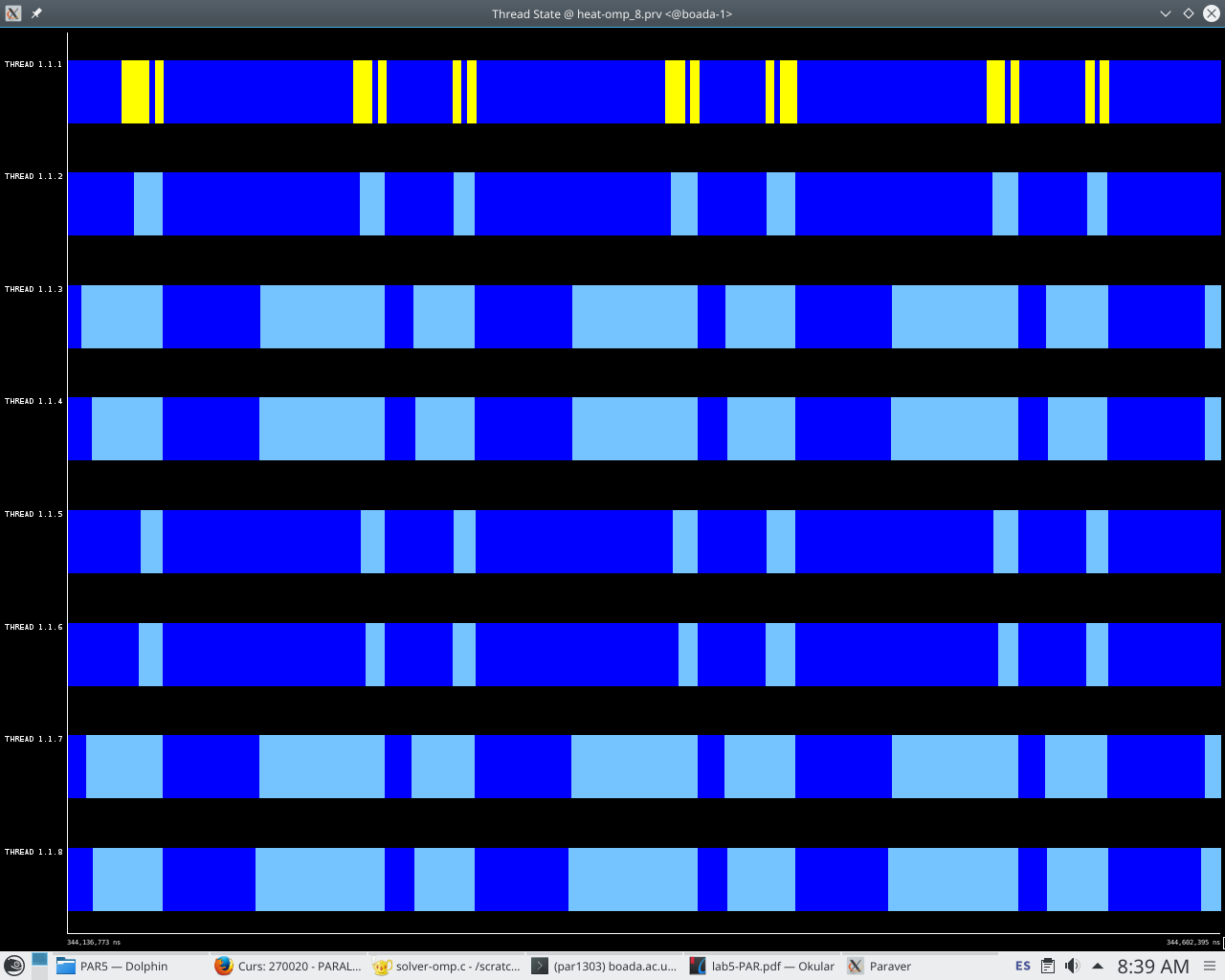


Figure 11: Jacobi trace with 8 threads

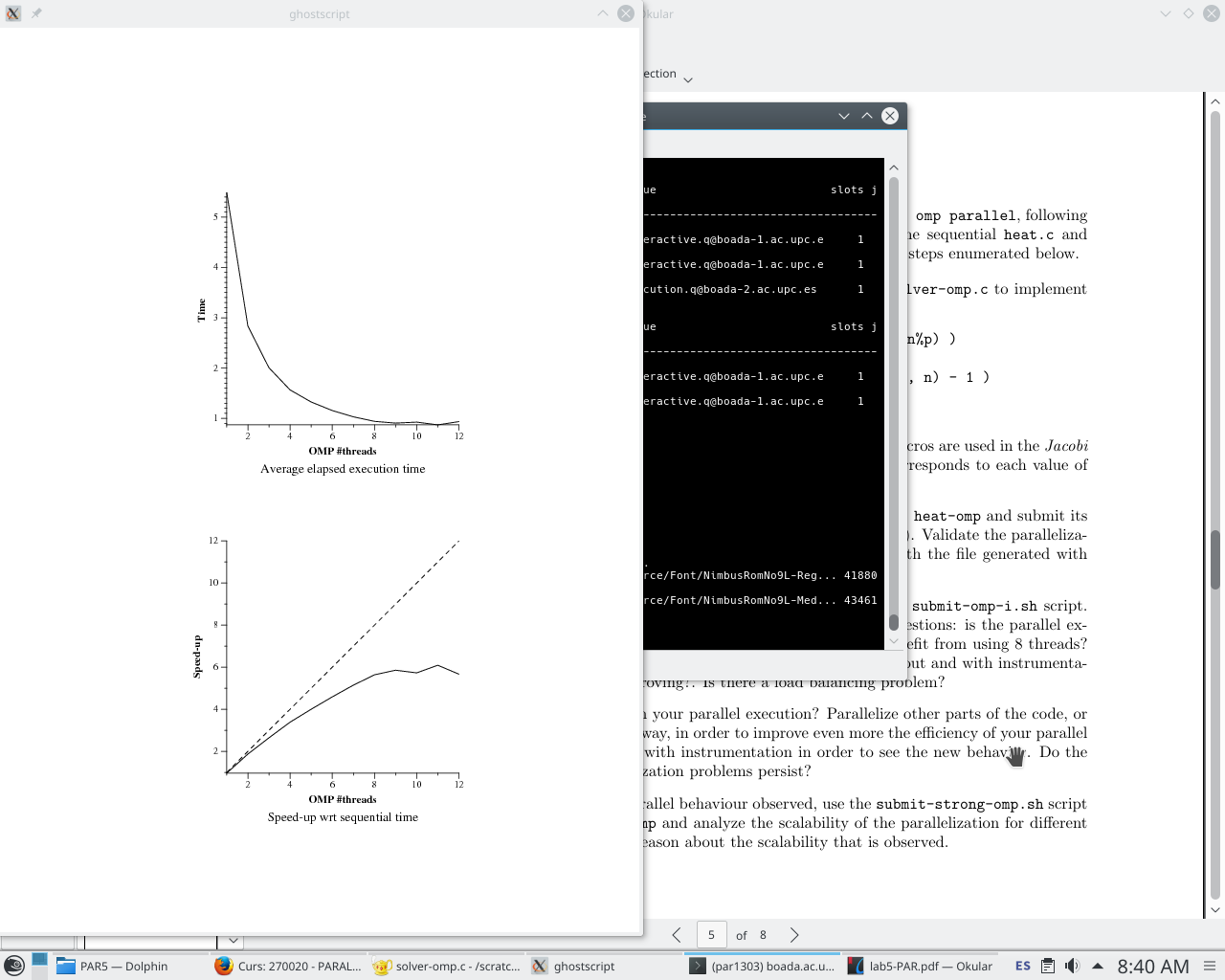
For the last part we can analyze the plots with the elapsed time and the speed-up:

Figure 12: Jacobi plots with 8 threads

We can see in the plot that the parallelization increases as the number of threads rise until we reach 10 threads. This is caused by the overhead result of synchronization processes.

**Part III: Parallelization of Gauss-Seidel with OpenMP**

We now parallelize the gauss-Seidel version. The modified code is shown in figure 13.

|  |
| --- |
| double relax\_gauss (double \*u, unsigned sizex, unsigned sizey)  {  double unew, diff, sum=0.0;  int howmany=omp\_get\_max\_threads();  int columnes=8;    #pragma omp parallel for ordered (2) private(unew, diff) reduction(+:sum)  for (int blockid = 0; blockid < howmany; ++blockid) {  for (int columna = 0; columna < columnes; ++columna) {  int i\_start = lowerb(blockid, howmany, sizex);  int i\_end = upperb(blockid, howmany, sizex);  int j\_start = lowerb(columna, howmany, sizey);  int j\_end = upperb(columna, howmany, sizey);  #pragma omp ordered depend(sink: blockid -1, columna) depend(sink:blockid, columna - 1)  for (int i=max(1, i\_start); i<= min(sizex-2, i\_end); i++) {  for (int j=max(1, j\_start); j<= min(sizey-2, j\_end); j++) {  unew= 0.25 \* ( u[ i\*sizey + (j-1) ]+ // left  u[ i\*sizey + (j+1) ]+ // right  u[ (i-1)\*sizey + j ]+ // top  u[ (i+1)\*sizey + j ]); // bottom  diff = unew - u[i\*sizey+ j];  sum += diff \* diff;  u[i\*sizey+j]=unew;  }  }  #pragma omp ordered depend(source)  }  }  return sum;  } |

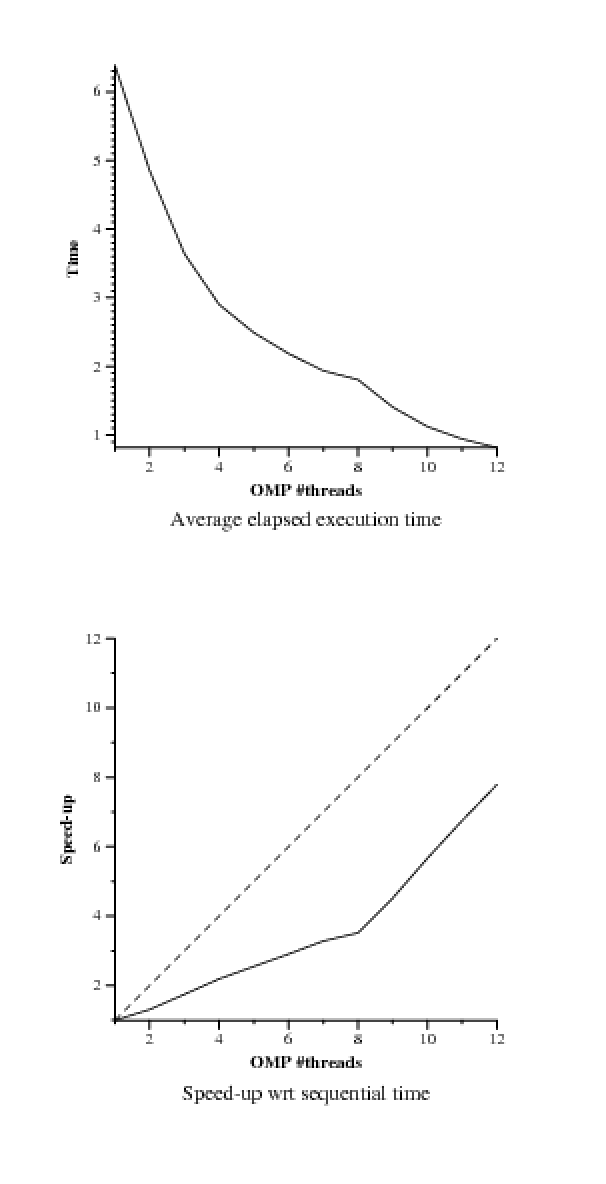
Figure 13: Gauss-Seidel code with #pragma omp ordered

When we execute this code with 8 threads we obtain the following trace (figure 14), and we get an execution time of 306,864 milliseconds.



Figure 14: Gauss-Seidel trace obtained with 8 threads

When checking the strong scalability, we can see the plots in figure 15.

Figure 15: Plots for Gauss version

We can see the no matter the amount of threads we have, the execution time continues to improve, so it has a very good strong scalability