Lab 5: Geometric (data) decomposition:

heat diffusion equation

*2018-2019 Q2*

**par2110**

Maurici Abad Gutierrez

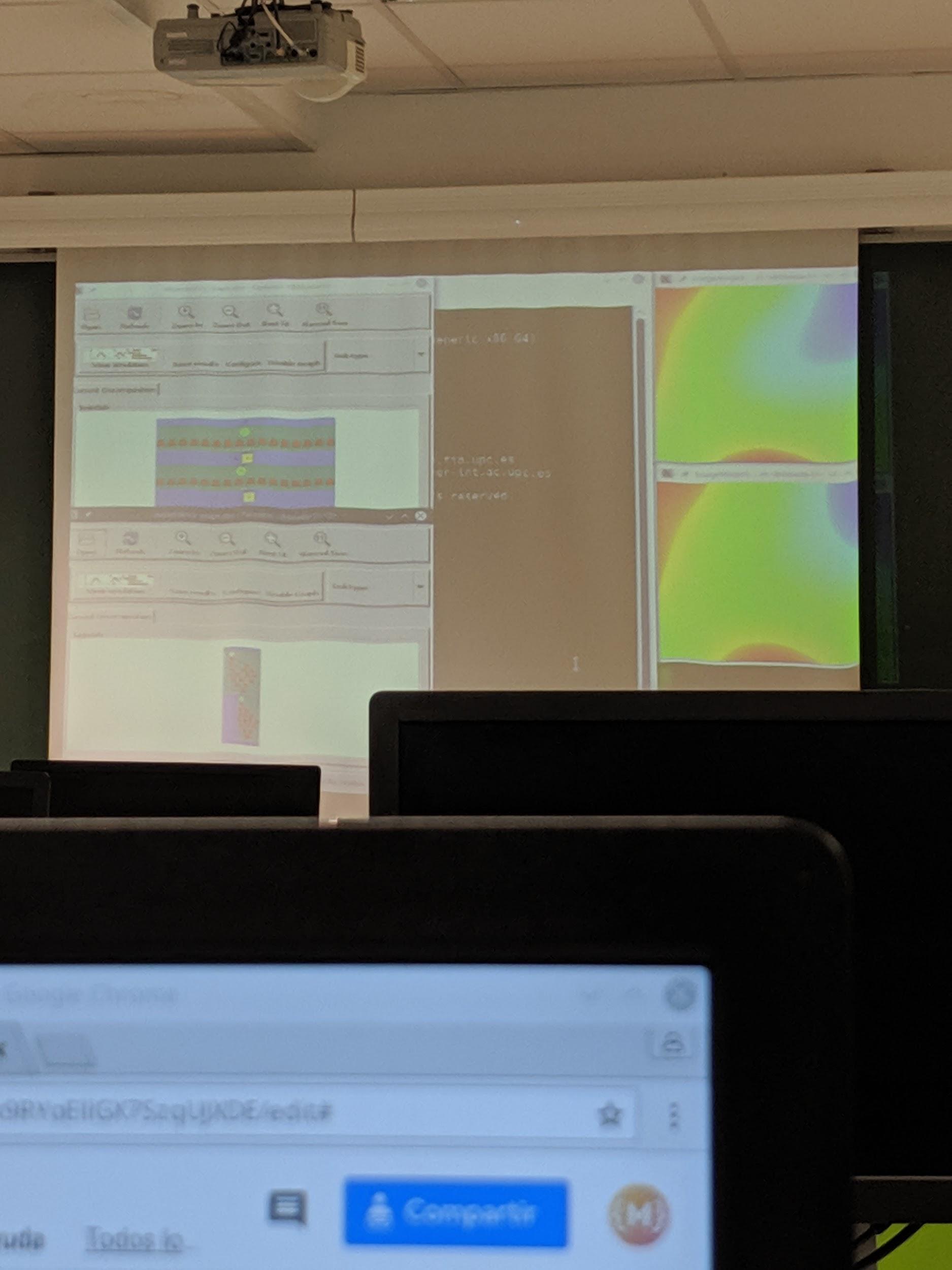
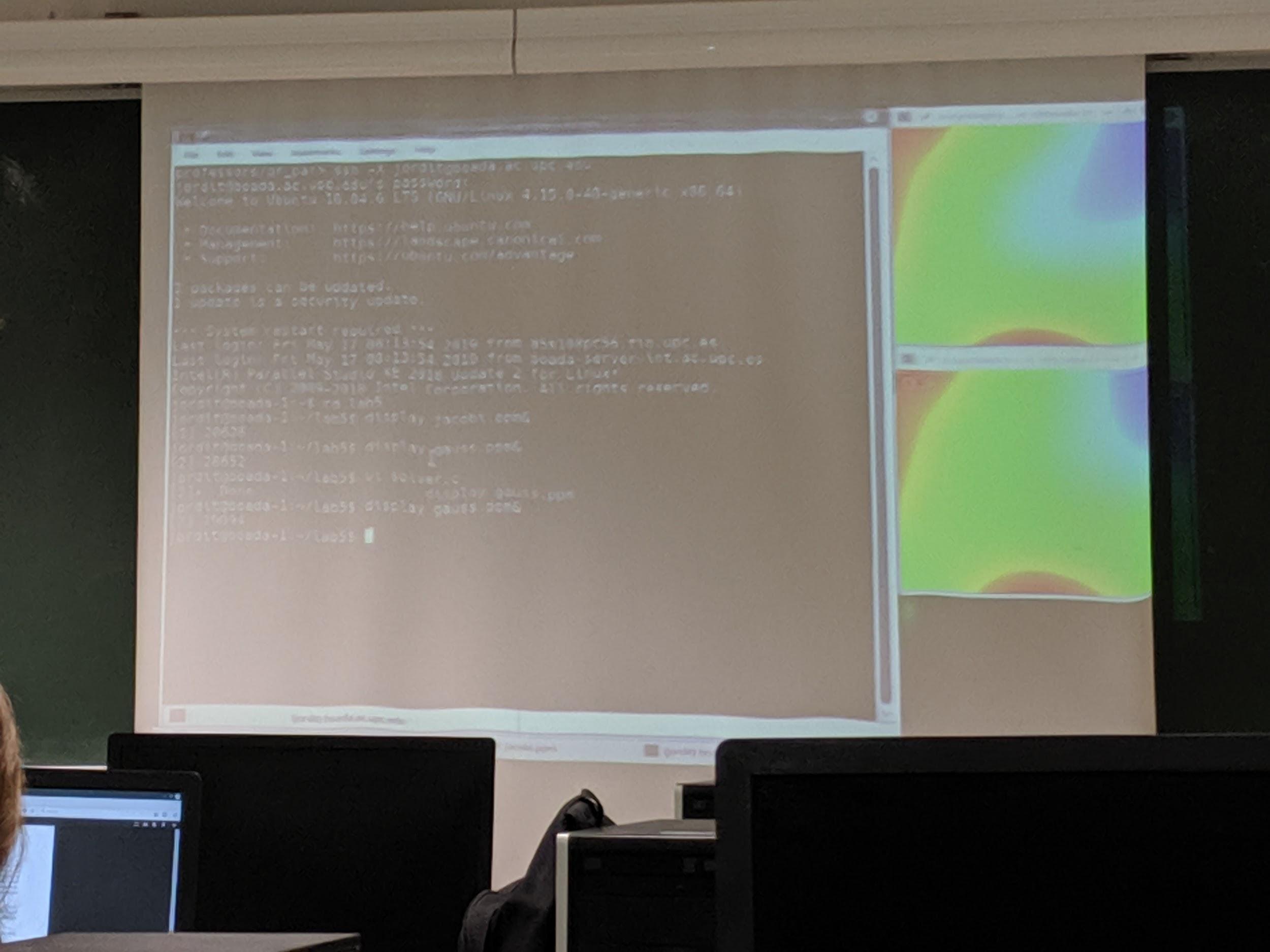
Nicolas Schrack

Lab1 -----------

1. Move code from heat to other file.

Tareador disable object () ← sum variable address

Remove sum from the analyisis, and we see that all iterations are in horitzontal.



Lab2 -----------

(explanation made in lab1)We have the sum dependency. To solve it we should use a **reduction**.

The variable dif has dependencies although tareador doesn’t show it. To solve it we should use a **local tasks for each variable**.

The overhead of the maximum granulatity causes to much overhead. The solution must group consecutive pixels.The solution is to group in blocks → **block\_size = n\_rows/n\_threads** , so each thread does a block.

Why we have three loops? Because the teachers prepared the code in order to solve it. We have to take adventage of the first loop. **We have to change the 4, use n\_threads instead**.

Then we have to change the work distribution (not with ~~schedule(static, 1)~~). **Use thread\_id and put a if** inside the for. He also said something of using “how\_many”.

4. Is there any serious serialization in your parallel execution? YES.   
We have to parallelize the matrix. (there is an alternative, but if we decide to use it we have to implement also the sequential version, and the teacher will put the same mark)

Lab2 -----------

Gauss

We have to do locking by column.

**We have to add a new loop** in gauss. Then implement the syncronization of the blocks. **Use structure from slide “The doacross loop nest (cont.)”**

For block\_col  
 For block\_row  
 For pixel\_col  
 For pixel\_row

depend(sink: i-1,j)

~~depend(sink: i,j-1)~~ we dont need this, because it’s implicit

LAB4-----------------

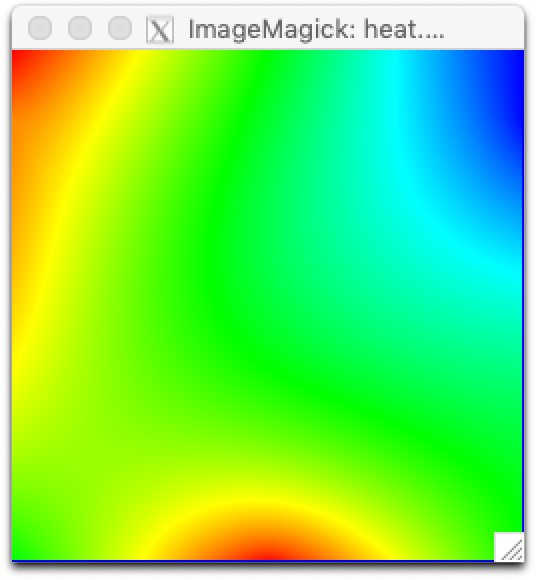
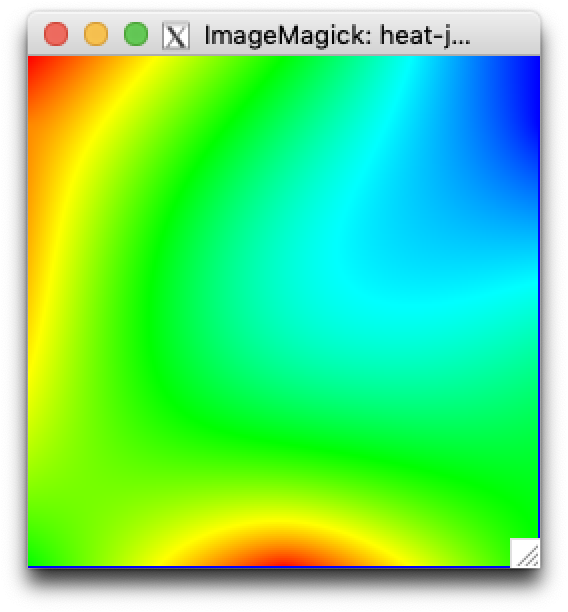
We have all the day to deliver it.

See slide 61 (The doacross loop nest) to resolve gauss.

# 1. Introduction

In this assignment we want to explore different parallisations of the sequential code heat.c. The code implements two algorithms to create an image of the heat distribution of an object. The solvers are Jacobi and Gauss-Seidl.

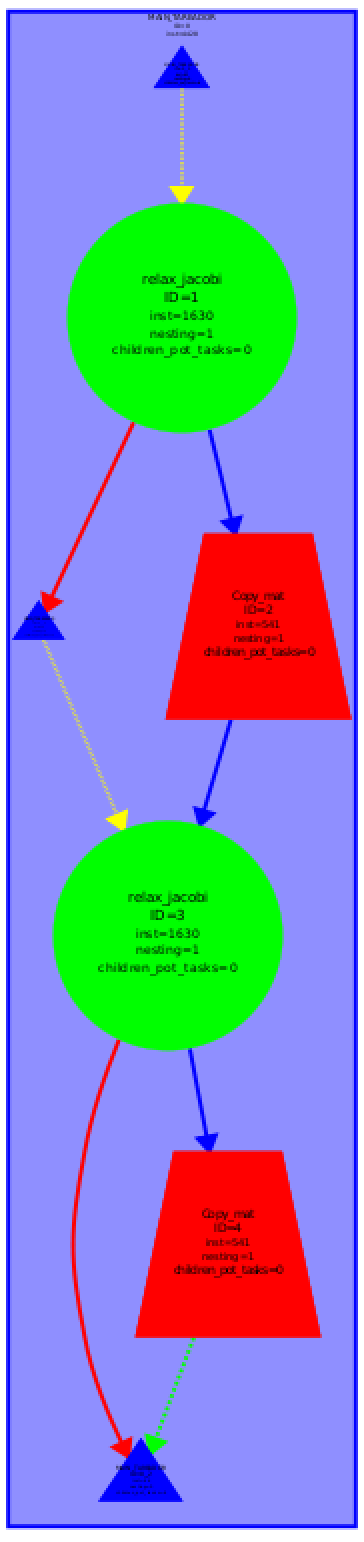
These are the images created with the different algorithms. The images are not the same because the Gauss-Seidl writes the calculated pixels right into the matrix which affect the other pixels, compared to the Jacobi.



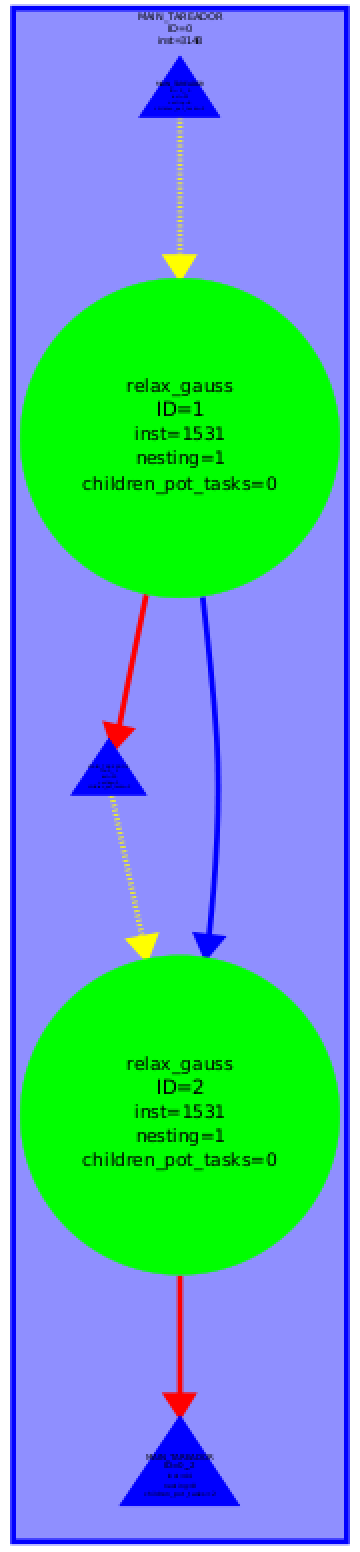
*Jacobi dependencies Gauss-Seidl*

# 2. Analysis with Tareador

In the Jacob version we see one more dependency than in Gauss’. That’s because Jacobi saves the points into another matrix, introducing one more dependency.



*Jacobi dependencies*



*Gauss dependencies*

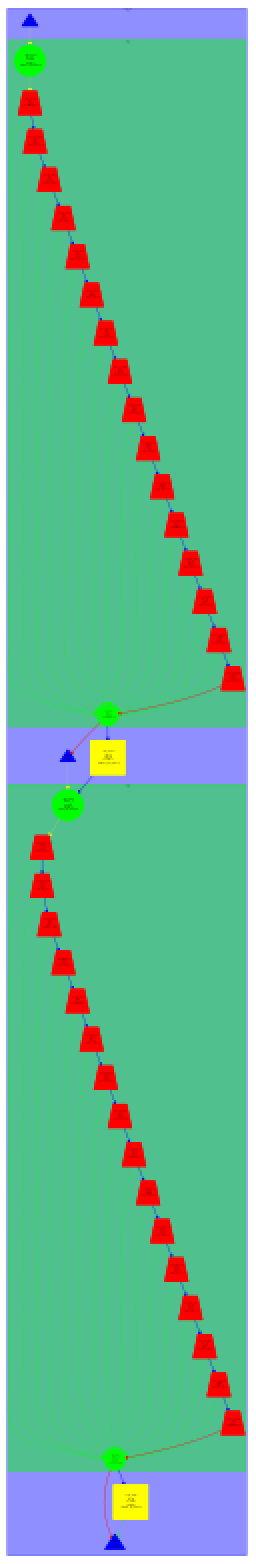
## Granulated dependency Graphs

In the following graphs we examine a finer granularity of the two algorithms. The granularity is the innermost loop in each of the solvers. The sum variable is causing the serialization of the tasks. If we ignore the sum variable from the analysis with tareador\_disable\_object(&sum); the dependency graph is completely flat (see *Jacobi dependencies more granulated and without sum*) in the Jacobi’s case. That means that sum is the only dependency in the program.

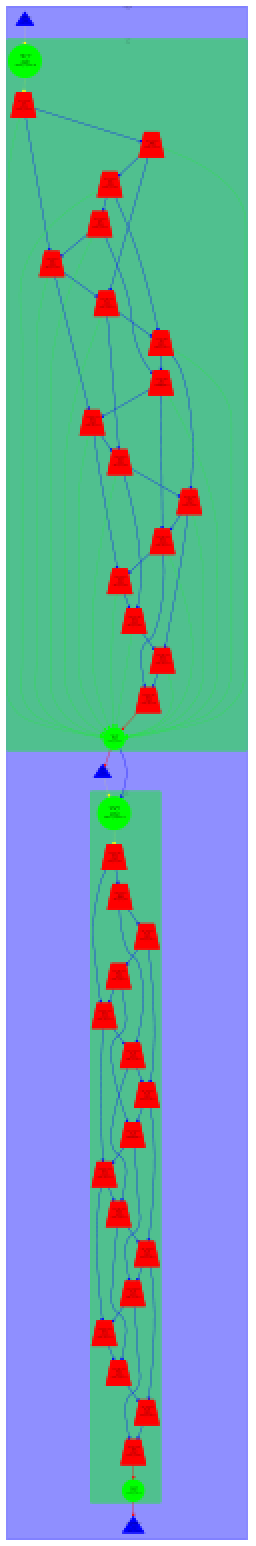
On the other hand, the Gauss dependency graph we can see that there are still some dependencies.

For the OpenMp implementation it will be important to share the sum variable between the threads, in order to prevent sequential execution. This could be done with a **reduction construct**.

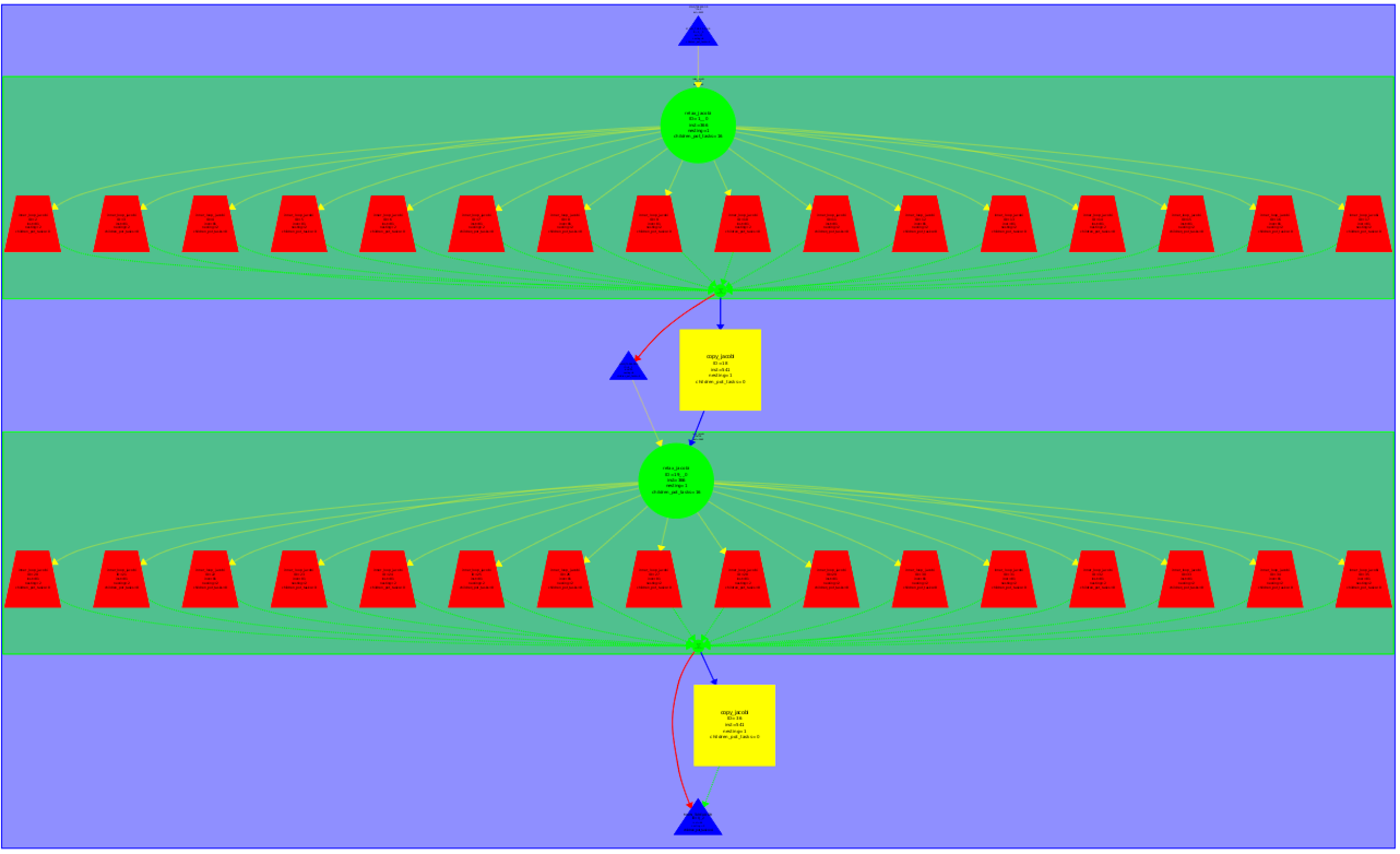
## 



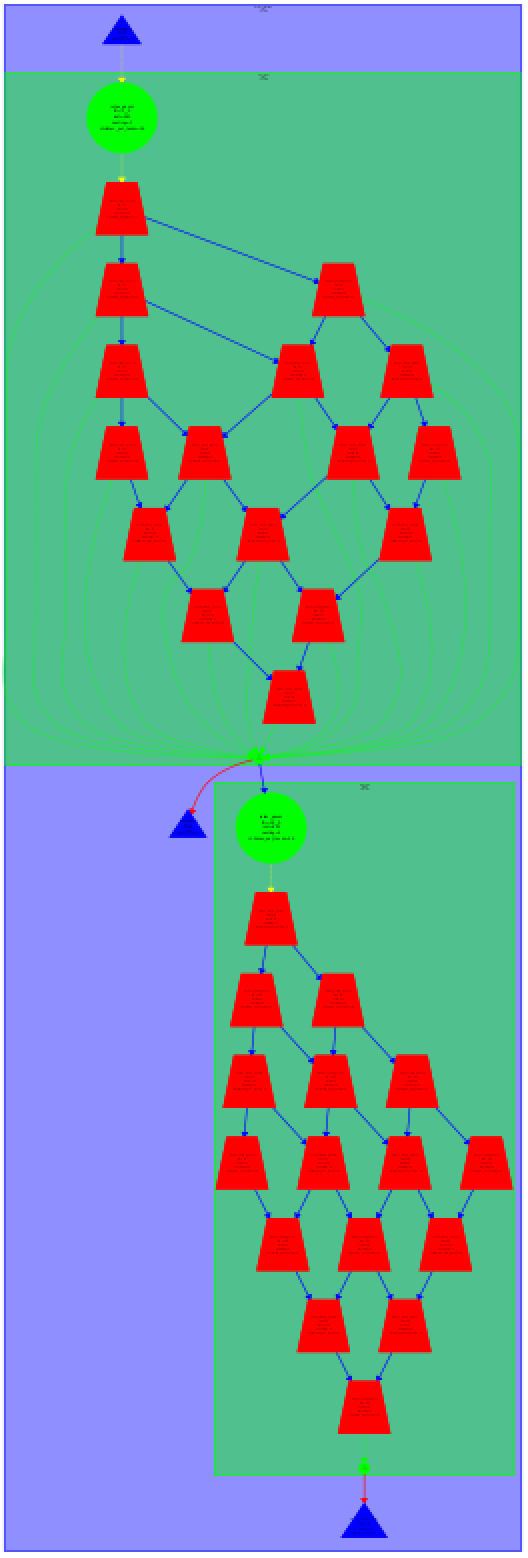
*Jacobi dependencies   
more granulated*



*Gauss dependencies   
more granulated*



*Jacobi dependencies   
more granulated and without sum*



*Gauss dependencies   
more granulated and without sum*

## Code

### Innermost loop of jacobi

|  |
| --- |
| for (int j=1; j<= sizey-2; j++) {  **tareador\_start\_task("inner\_loop\_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\_loop\_jacobi");**  } |

### Innermost loop of gauss

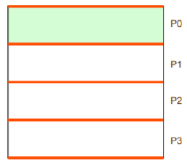
|  |
| --- |
| for (int j=1; j<= sizey-2; j++) {  **tareador\_start\_task("inner\_loop\_gauss");**  unew= 0.25 \* ( u[ i\*sizey + (j-1) ]+ // left  u[ i\*sizey + (j+1) ]+ // right  u[ (i-1)\*sizey + j ]+ // top  u[ (i+1)\*sizey + j ]); // bottom  diff = unew - u[i\*sizey+ j];  **tareador\_disable\_object(&sum);**  sum += diff \* diff;  **tareador\_enable\_object(&sum);**  u[i\*sizey+j]=unew;  **tareador\_end\_task("inner\_loop\_gauss");**  } |

# 3. Parallelization of Jacobi with OpenMP parallel

## Code parallelised of Jacobi

|  |
| --- |
| **void** **copy\_mat** (**double** \*u, **double** \*v, **unsigned** sizex, **unsigned** sizey) {  #pragma omp parallel for collapse(2) schedule(static,1)  **for** (**int** i=1; i<=sizex-2; i++)  **for** (**int** j=1; j<=sizey-2; j++)   v[ i\*sizey+j ] = u[ i\*sizey+j ]; }  **double** **relax\_jacobi** (**double** \*u, **double** \*utmp, **unsigned** sizex, **unsigned** sizey) {  **double** diff, sum=0.0;  #pragma omp parallel private (diff) reduction(+: sum)  {  **int** blockid = omp\_get\_thread\_num();  **int** howmany=omp\_get\_num\_threads();  **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; } |

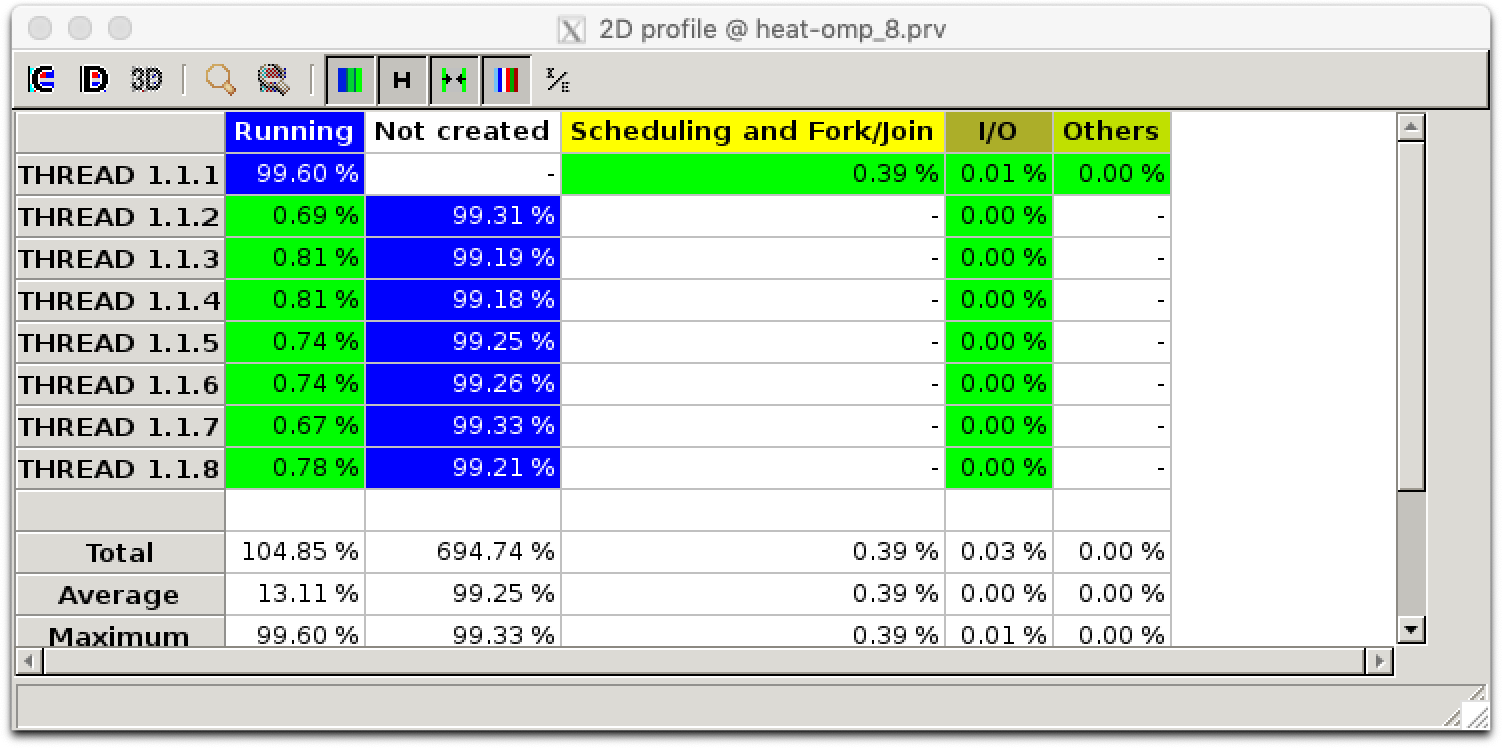
## Geometric data decomposition.



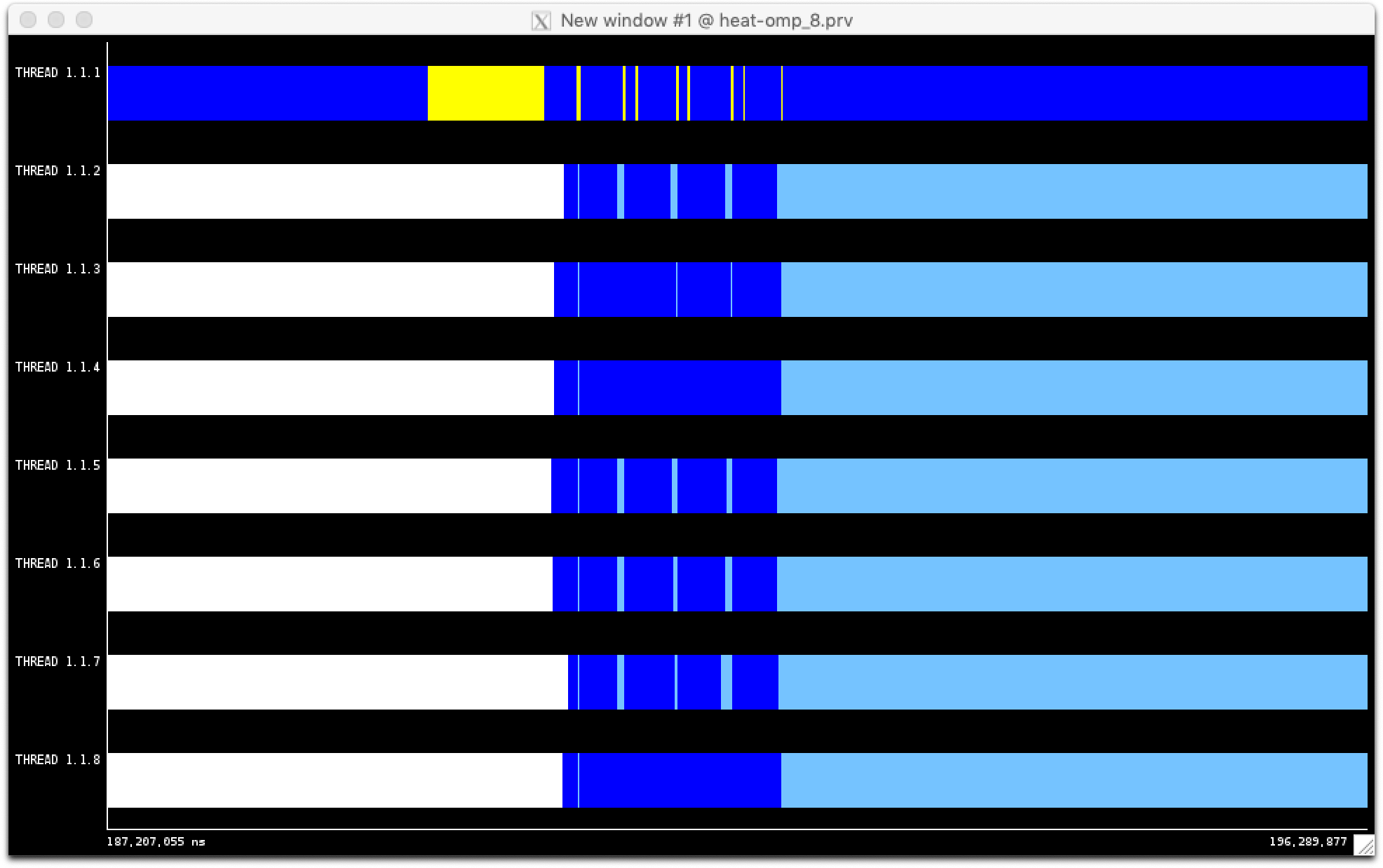
Data decomposition for Jacobi

## Explanation

Although it seems that the first thread is doing all the work, it’s actually doing the initialization of the program. And the other threads do the important part of the program. To visualize that in paraver, you have to zoom in.



Running time per thread

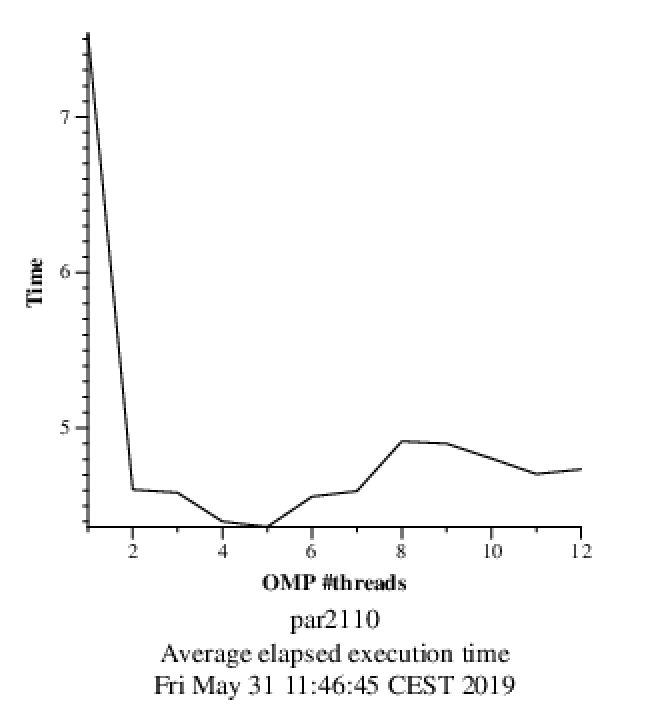
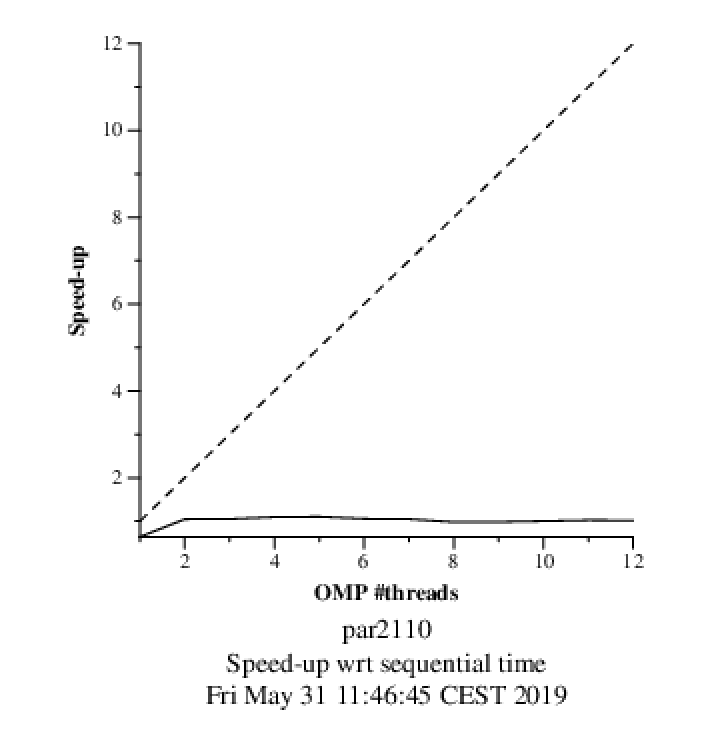


Zoomed in trace

After examining the the code again, we found other parts that could be parallelized.

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

The parallel behaviour of the execution with different number of processors is the following.



Execution time Speed-up to sequential

The speed-up doesn’t increase with the increase of the numbers of threads after 2. The execution time is the shortest at around 4 and increases after that.

# 

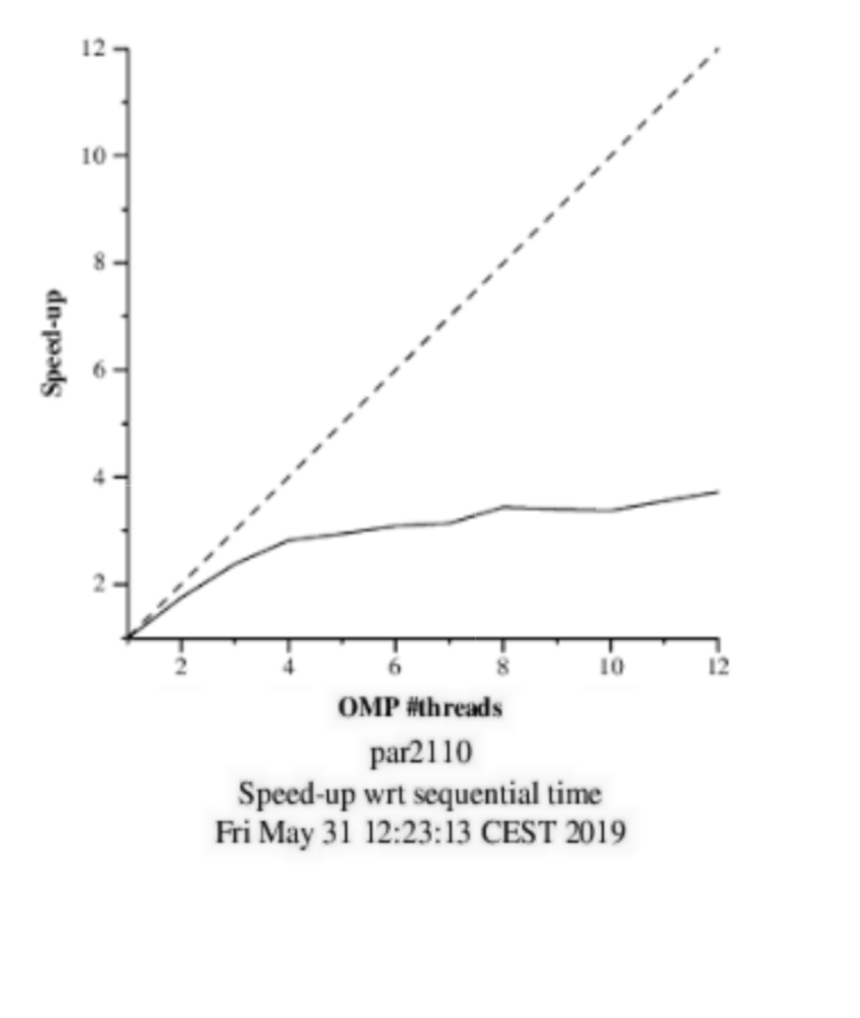
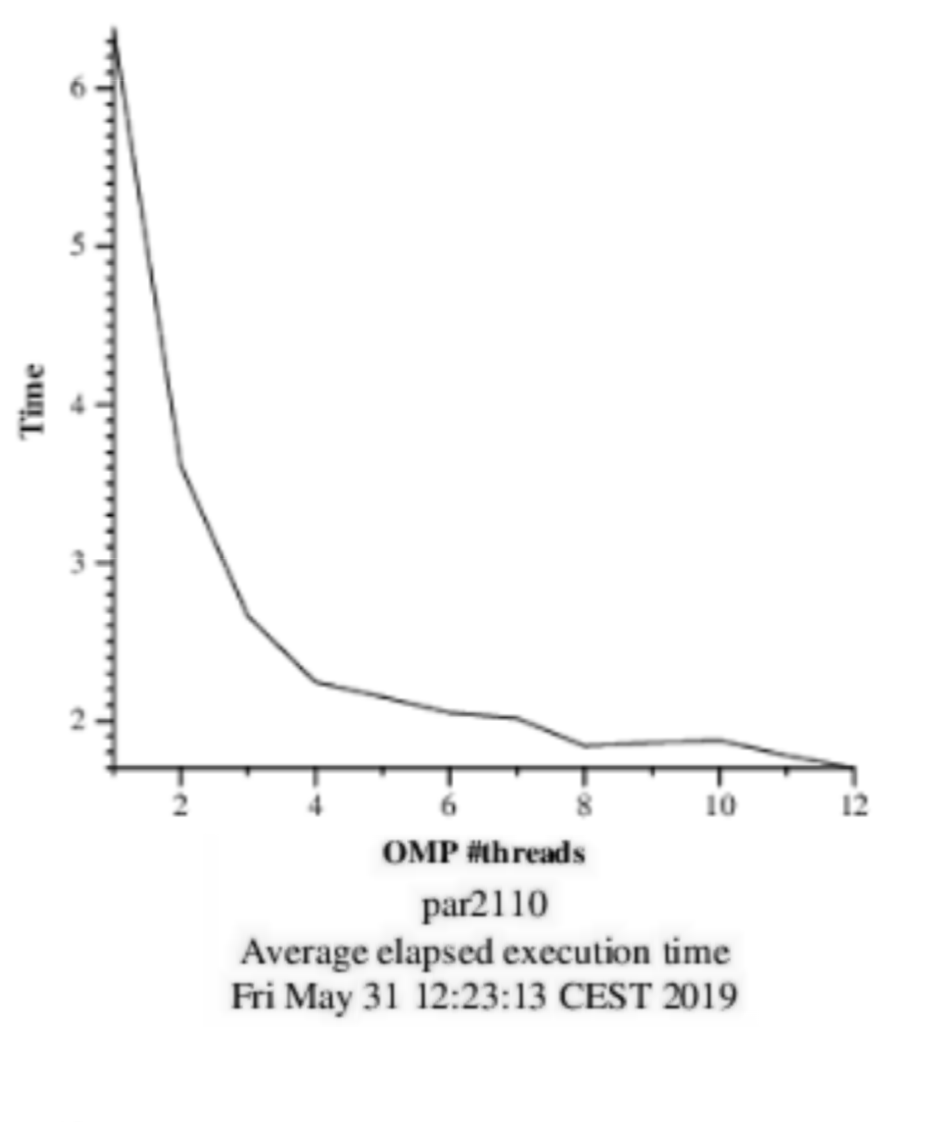
# 4. Parallelization of Gauss-Seidel with OpenMP ordered

|  |
| --- |
| **double** **relax\_gauss** (**double** \*u, **unsigned** sizex, **unsigned** sizey)  {  **double** unew, diff, sum=0.0;  **int** howmany=4;   #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)  **for** (**int** blockid\_row = 0; blockid\_row < howmany; ++blockid\_row) {  **for** (**int** blockid\_col = 0; blockid\_col < howmany; ++blockid\_col) {  **int** i\_start = lowerb(blockid\_row, howmany, sizex);  **int** i\_end = upperb(blockid\_row, howmany, sizex);  **int** j\_start = lowerb(blockid\_col, howmany, sizey);  **int** j\_end = upperb(blockid\_col, howmany, sizey);   #pragma omp ordered depend (sink: blockid\_row-1, blockid\_col)  **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; } |

The code runs as expected. We used the doacross loop nest structure.



Trace of Parallelized Gauss

By taking a look on the paraver trace and the Scalability graphics we can conclude that at list is much better than Jacobi one.

We also can conclude that the optimum value of threads is 12, because the execution time decreases with the amount of threads steadily.