Parallelism

Lab 3

Gerard Bayego

Martín Dans

Grup: par3101

Curs: 2014-15

Fall semester

**Question 5.1.1:**

**Tareador API:**

*Jacobi:*

In the file solver-tareador.c we added, in the function relax\_jacobi, inside the inner for:

tareador\_start\_task("Internj");

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("Internj");

*Gauss-Seidel:*

In the file solver-tareador.c we added, in the function relax\_gauss, inside the inner for:

tareador\_start\_task("Interng");

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];

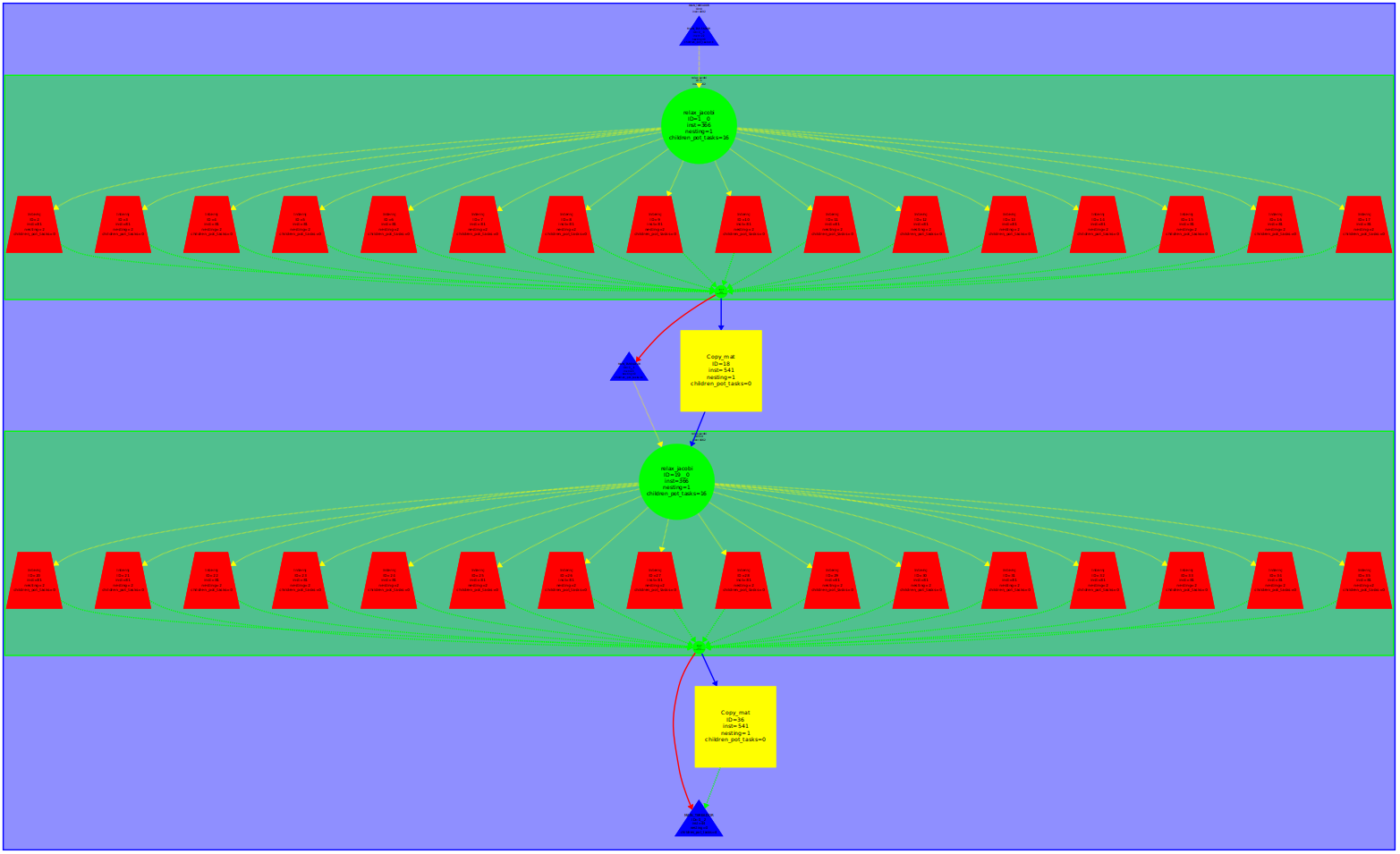
tareador\_disable\_object(&sum);

sum += diff \* diff;

u[i\*sizey+j]=unew;

tareador\_end\_task("Interng");

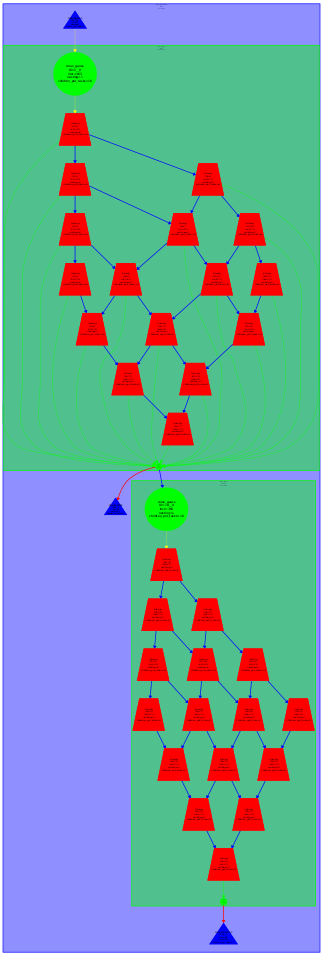
***Jacobi:***

The problem is that the variable sum where they store the partial summations is shared. This variable is accessed by every thread and that makes to tareador think that it is a dependency. Of course that’s not our case since we only want to sum and the order doesn’t matter.

We have to notice that even if it’s parallelizable reading the variable and writing on it could produce a race condition between threads. To prevent that we can make that region critical, reduction(+:sum) or do partial summations for every thread and add all of them at the end.

Also copy\_mat can be paralyzed, with a simple pragma omp for collapse 2, because didn’t have any dependence.

***Gauss-Seidel:***



In the gauss case, each position depends of the element upside and left side, and this generates a lot of dependences.

When we parallelize with blocks, first we can compute block 0-0, after, we can compute 0-1 and 1-0 blocks. This parallelization creates a tree of dependences, that first increase the number of tasks to do, and after decrease to the end.

**Question 5.2.1:**

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 reduction(+:sum)

{

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++) {

#pragma omp for nowait private(diff)

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;

}

To parallelize jacobi, we added a omp parallel pragma with a reduction of sum variable, to avoid the dependence detected in the tareador, also we added a pragma omp for to create tasks of work of the most inner for of the function.

void copy\_mat (double \*u, double \*v, unsigned sizex, unsigned sizey) {

#pragma omp parallel for collapse(2)

for (int i=1; i<= sizex-2; i++){

for (int j=1; j<= sizey-2; j++){

v[ i\*sizey+j ] = u[ i\*sizey+j ];

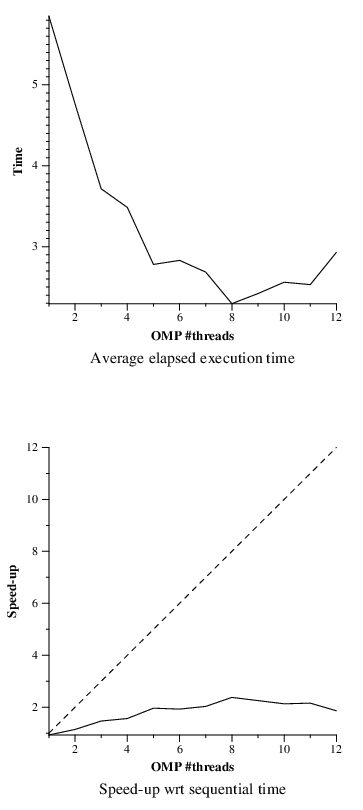
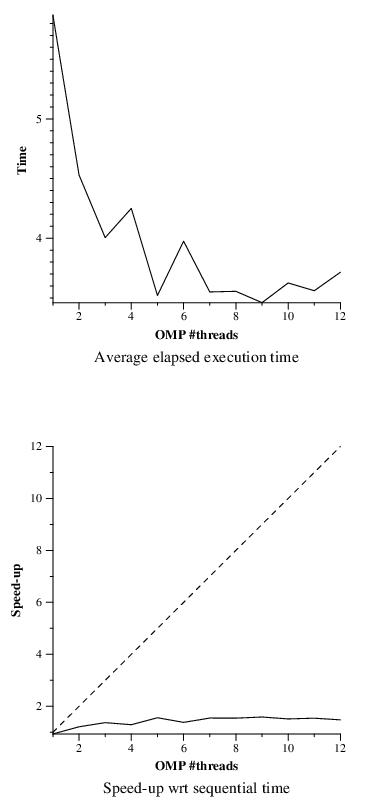
}

}

}

We also parallelized the copy mat with a simple omp for because no dependences exists, with collapse(2) to generate the max number of tasks possible and increase parallelism.

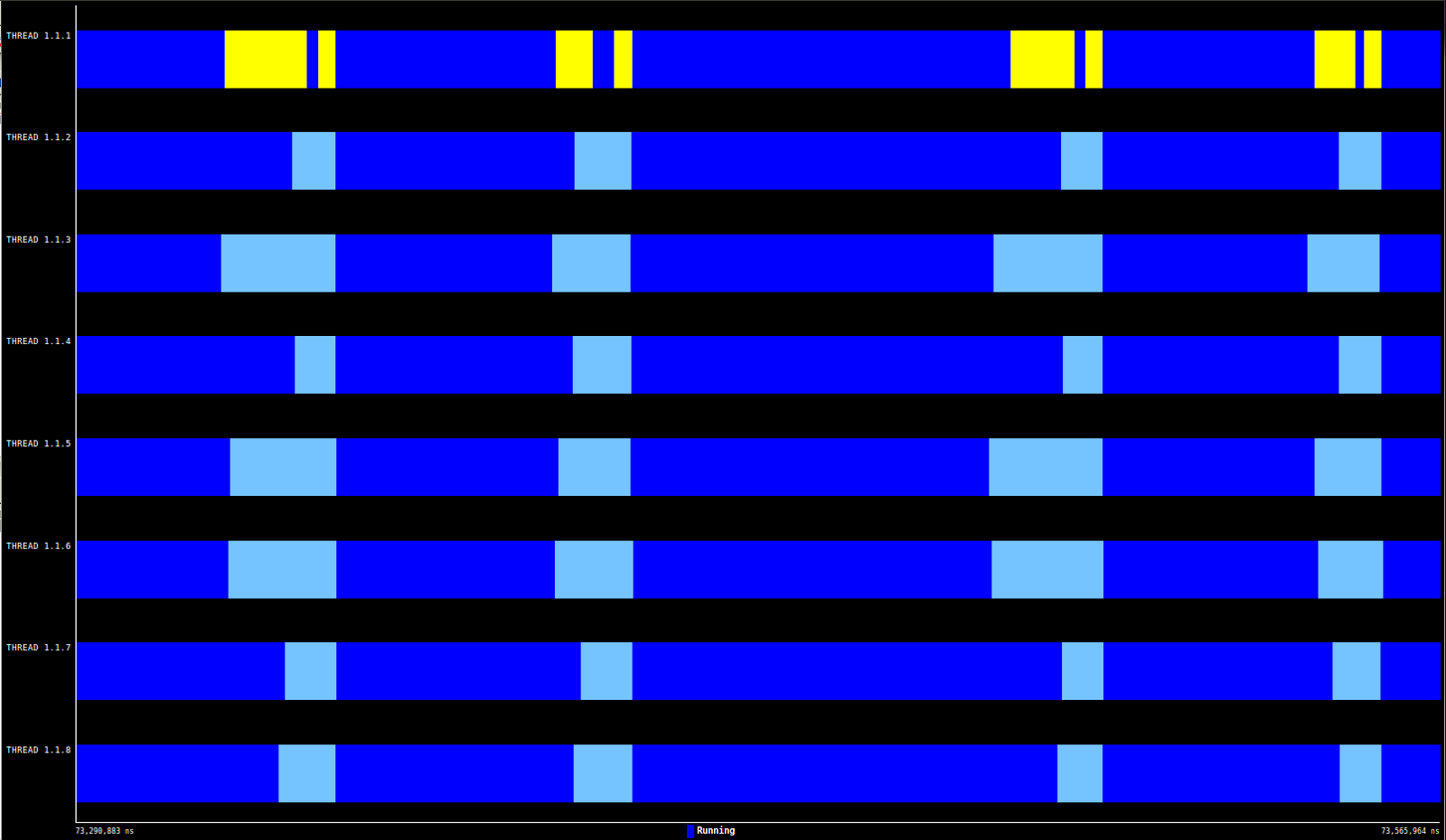
**Question 5.2.2:**

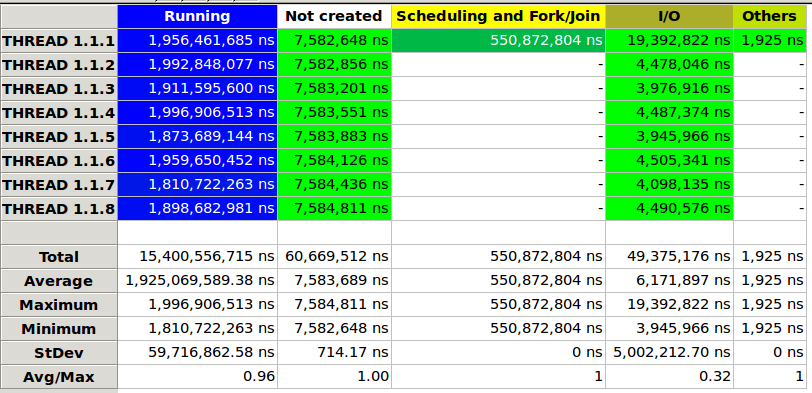


Plot before copy-mat parallelization Plot after copy-mat parallelization

At left we have the speedup with only de relax function parallelized. We can see that the speedup obtained it’s about 1.3. At right we can see the speed-up achieving 2 with 8 threads and then it starts decaying because the winnable time with the parallelization decays and the overhead increase.

With the copy-mat parallelization, we increase the speedup, because is fully parallelizable and needs small synchronization between tasks.





As we can see in the two captures of paraver, we have a lot of overhead caused for the synchronization and creation of tasks, that reduces the total speedup of the program.

**Question 5.3.1:**

double relax\_gauss (double \*u, unsigned sizex, unsigned sizey) {

double unew, diff, sum = 0.0;

int howmany = omp\_get\_max\_threads();

int numBlocs = 8;

int blocsProcesats[howmany];

for(int i = 0; i < howmany; ++i) blocsProcesats[i] = 0;

#pragma omp parallel for schedule(static) private(diff,unew) reduction(+: sum)

for(int i = 0; i < howmany; i++){

int ii\_start = lowerb(i,howmany,sizex);

int ii\_end = upperb(i,howmany,sizex);

for (int j = 0; j < numBlocs; j++){

int jj\_start = lowerb(j,numBlocs,sizey);

int jj\_end = upperb(j,numBlocs,sizey);

if(i > 0){

while(blocsProcesats[i-1]<=j){

#pragma omp flush

}

}

for(int ii = max(1,ii\_start); ii<= min(sizex-2, ii\_end); ii++){

for(int jj= max(1,jj\_start); jj<= min(sizey-2, jj\_end); jj++){

unew = 0.25\* (u[ii \* sizey + (jj-1)] + // left

u[ii \* sizey + (jj+1)] + // right

u[(ii-1) \* sizey + jj] + // top

u[(ii+1) \* sizey + jj]); // bottom

diff = unew - u[ii \* sizey + jj];

sum+= diff\*diff;

u[ii\*sizey + jj] = unew;

}

}

++blocsProcesats[i];

#pragma omp flush

}

}

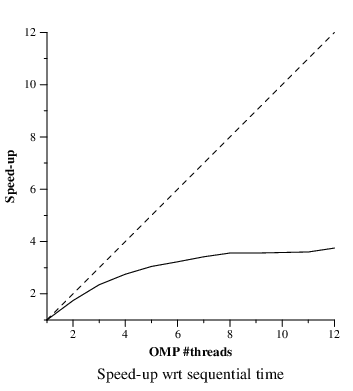
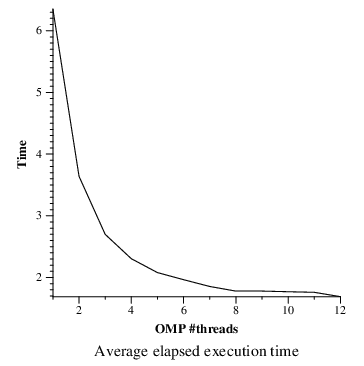
return sum;

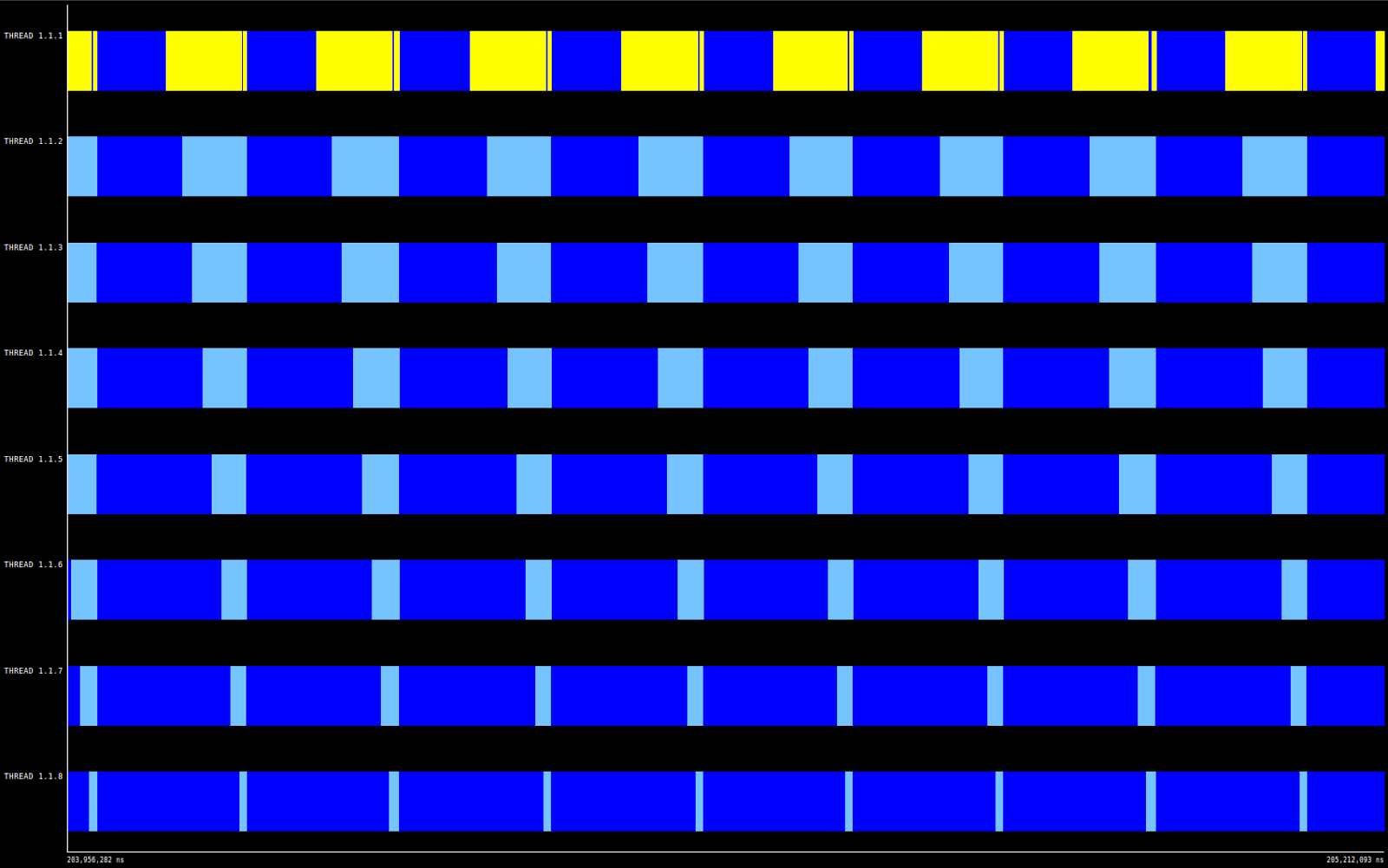
}

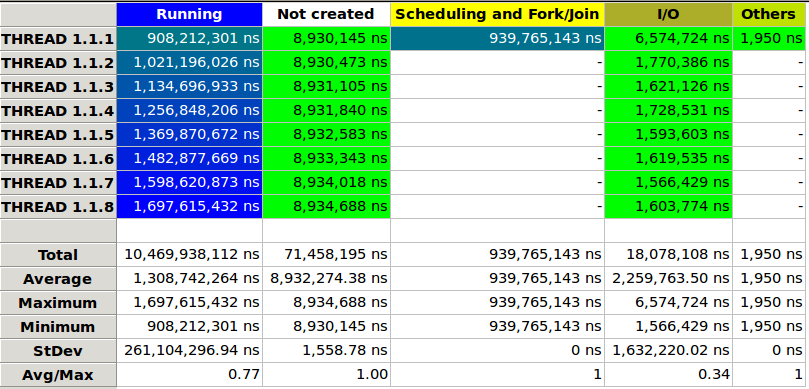
For this algorithm, the parallelization strategy is more complex than Jacobi, caused for the complex dependences.

We have to divide the matrix to give equally rows to all threads, after we are going to divide the matrix space of each thread in blocks. With this decomposition we have secured the dependence left to right, but we have to secure also the up dependence. To do this we created the blocsProcesats vector of the size of num\_threads. Here with a pooling strategy of pooling we wait the dependences needed, doing flush.

**Question 5.3.2:**

As we can see gauss has a better speedup compared to jacobby, but is not a big increase. The increase is produced because is more efficient the gauss version, because we didn’t need to copy matrix, and we save in the overhead of the program. But has more computation the gauss version.

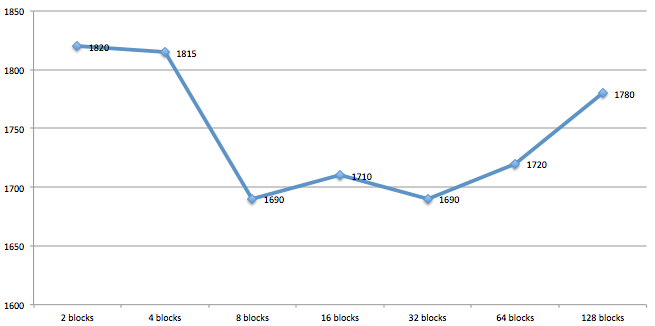




As we can see if we compare with the gauss profile, the synchronization time is less, and the working time is a little bit larger, and with this two things we can justify the small speedup.

**Question 5.3.2:**

In the gauss algorithm is so important to find the point with the optimum ratio between computation and synchronization. To do this we fixed the executions with 8 threads and we do various executions with different parameters.

We decided to do executions with 2, 4, 8, 16, 32, 64, 128 blocks, to see the effect of computation and synchronization (blue line is the execution time).

With the results we can see that the optimal value is between 8 and 32 blocks, because has the smaller execution times. With small values there are a lot of computation per thread and with large values we have a lot of synchronization time.