

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:

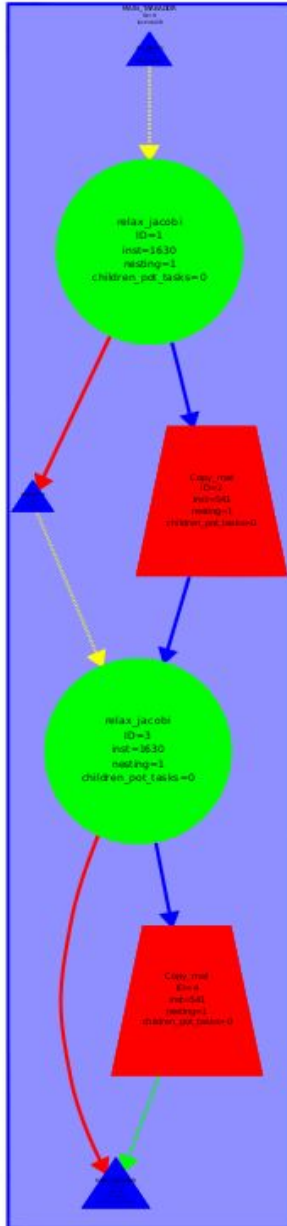


Figure 1: Jacobi tareador execution without tasks.

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 `"taredor_start_task("inner_jacobi");"` & `"taredor_end_task("inner_jacobi");"` between the inner for. The result is this:

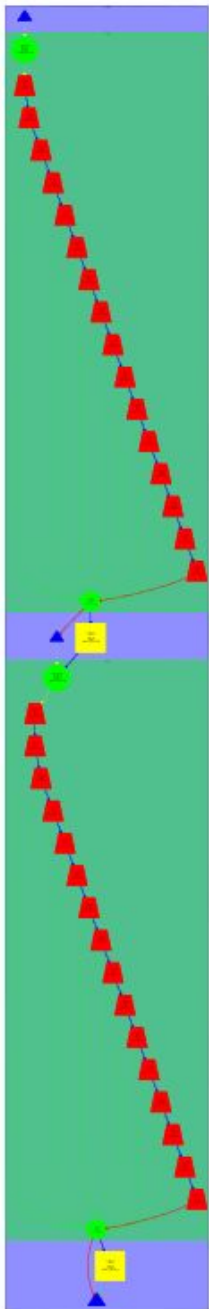


Figure 2: Jacobi with inner tasks defined (Taredor)

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:

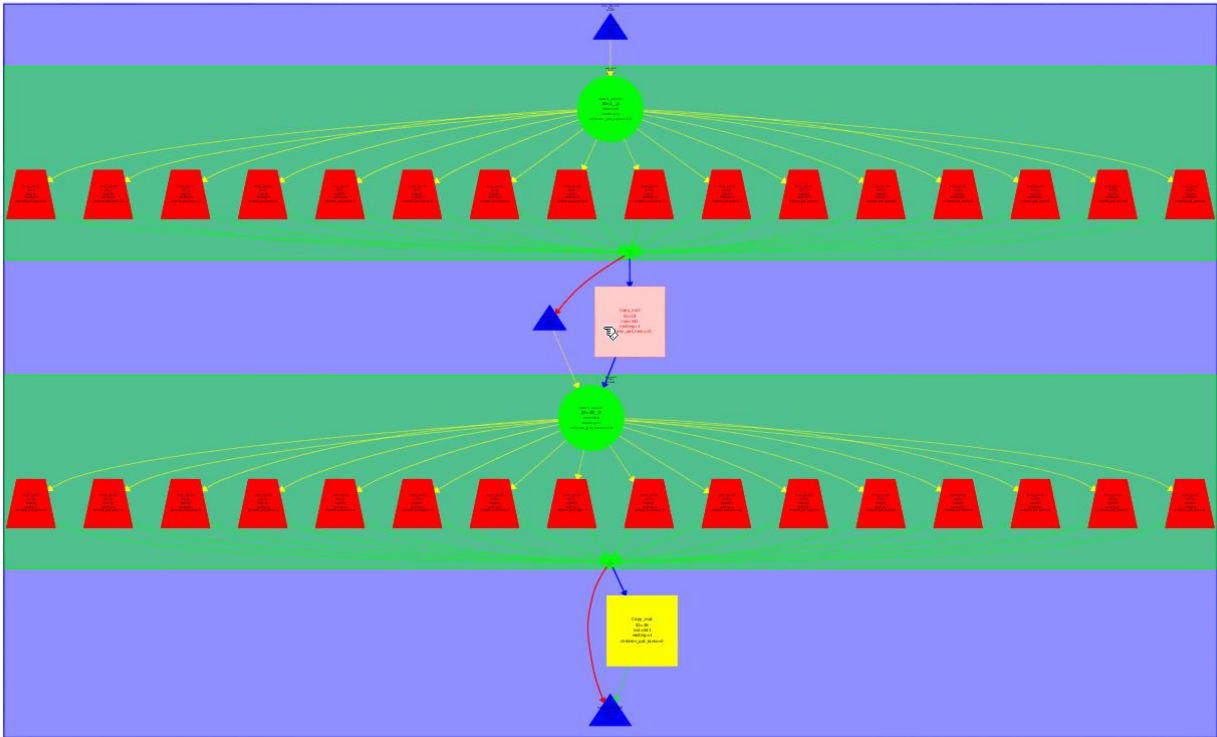


Figure 3: Jacobi with disable_object(&sum) (Tareador)

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:

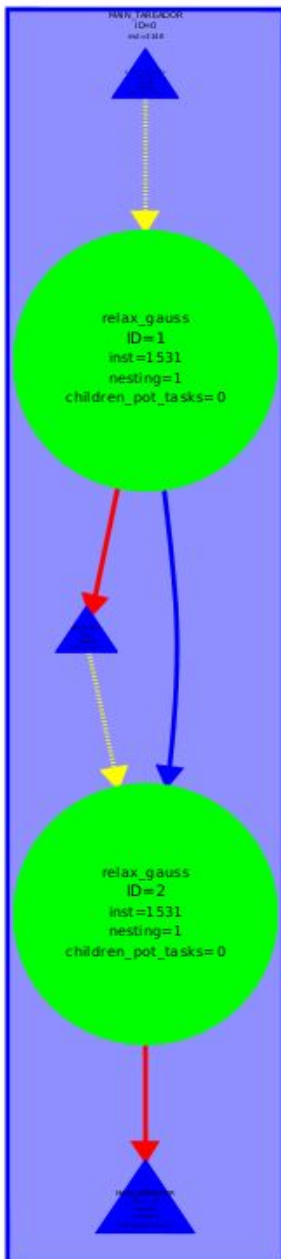


Figure 4: Gauss-Seidel tareador execution without tasks.

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:

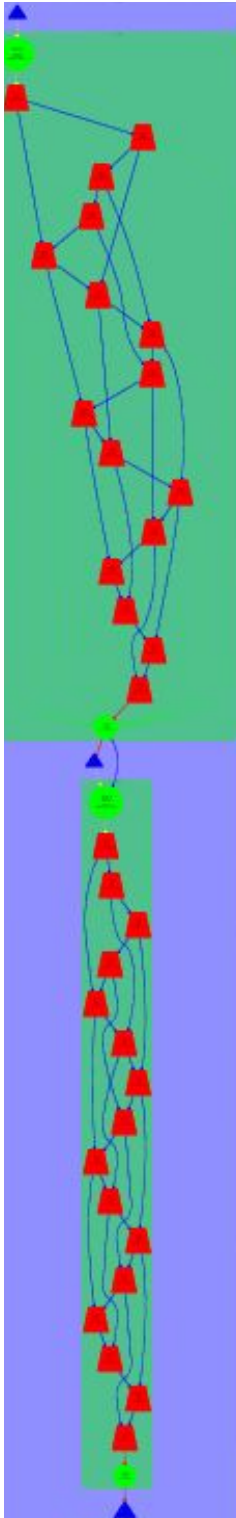


Figure 5: Gauss-Seidel with inner tasks defined (Tareador)

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:

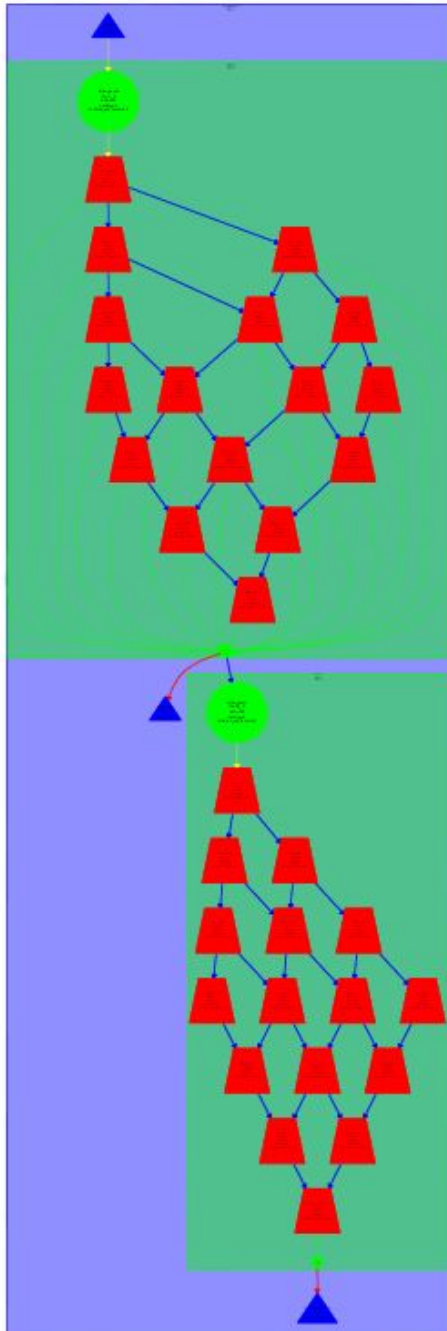


Figure 6: Gauss with `disable_object(&sum)` (Tareador)

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 taredor are the following:

Jacobi:

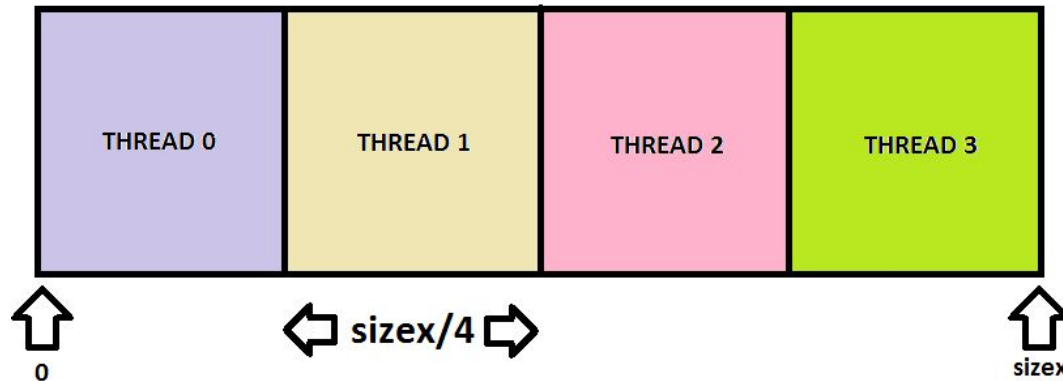
# threads	Temps (microseconds)
1	4.492
2	2.920
3	2.434
4	2.150
6	1.948
8	1.948

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.



BLOCK DISTRIBUTION

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 taredor 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 sizeof, unsigned sizeof)
{
    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, sizeof);
        int i_end = upperb(blockid, howmany, sizeof);
        for (int i=max(1, i_start); i<= min(sizeof-2, i_end); i++) {
            for (int j=1; j<= sizeof-2; j++) {
                utmp[i*sizeof+j]= 0.25 * ( u[ i*sizeof   + (j-1) ]+ // left
                                           u[ i*sizeof   + (j+1) ]+ // right
                                           u[ (i-1)*sizeof + j   ]+ // top
                                           u[ (i+1)*sizeof + j   ]); // bottom

                diff = utmp[i*sizeof+j] - u[i*sizeof + j];
                sum += diff * diff;
            }
        }
    }

    return sum;
}
```

Figure 8: Jacobi code with `#pragma omp parallel`

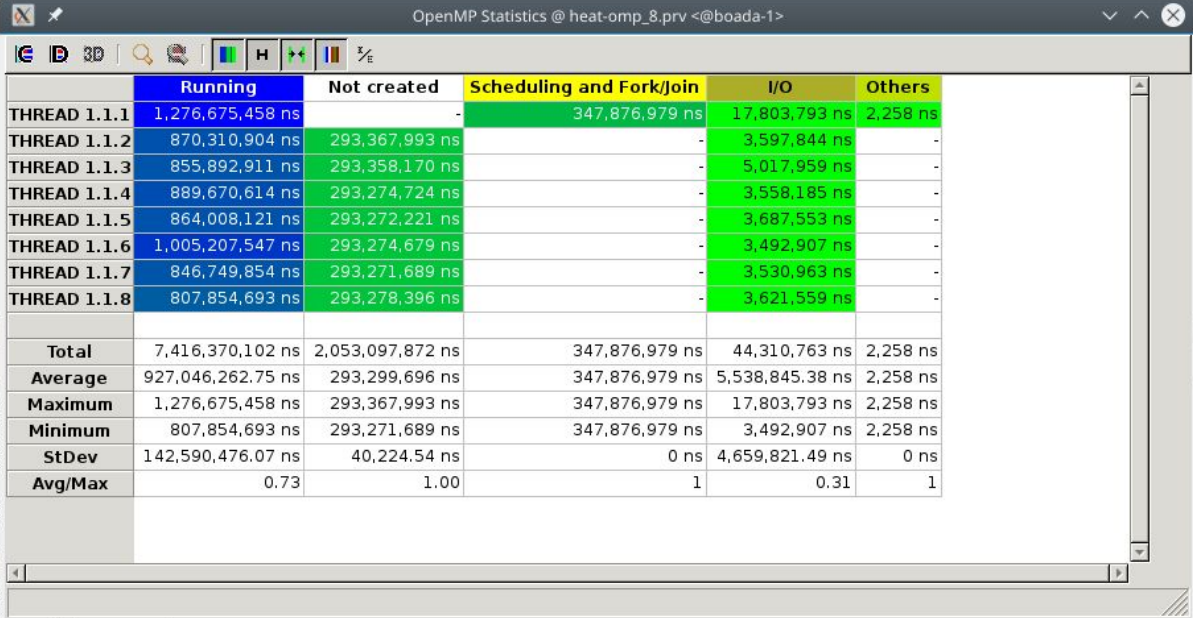
Also we have changed the value for *howmany* because the old value was 4. That value didn't benefit for the 8 or more threads value, so the execution time was increased for that reason. So we changed the value for `omp_get_max_threads()` so we can benefit for the maximum values of threads.

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:



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,276,675,458 ns	-	347,876,979 ns	17,803,793 ns	2,258 ns
THREAD 1.1.2	870,310,904 ns	293,367,993 ns	-	3,597,844 ns	-
THREAD 1.1.3	855,892,911 ns	293,358,170 ns	-	5,017,959 ns	-
THREAD 1.1.4	889,670,614 ns	293,274,724 ns	-	3,558,185 ns	-
THREAD 1.1.5	864,008,121 ns	293,272,221 ns	-	3,687,553 ns	-
THREAD 1.1.6	1,005,207,547 ns	293,274,679 ns	-	3,492,907 ns	-
THREAD 1.1.7	846,749,854 ns	293,271,689 ns	-	3,530,963 ns	-
THREAD 1.1.8	807,854,693 ns	293,278,396 ns	-	3,621,559 ns	-
Total	7,416,370,102 ns	2,053,097,872 ns	347,876,979 ns	44,310,763 ns	2,258 ns
Average	927,046,262.75 ns	293,299,696 ns	347,876,979 ns	5,538,845.38 ns	2,258 ns
Maximum	1,276,675,458 ns	293,367,993 ns	347,876,979 ns	17,803,793 ns	2,258 ns
Minimum	807,854,693 ns	293,271,689 ns	347,876,979 ns	3,492,907 ns	2,258 ns
StDev	142,590,476.07 ns	40,224.54 ns	0 ns	4,659,821.49 ns	0 ns
Avg/Max	0.73	1.00	1	0.31	1

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.

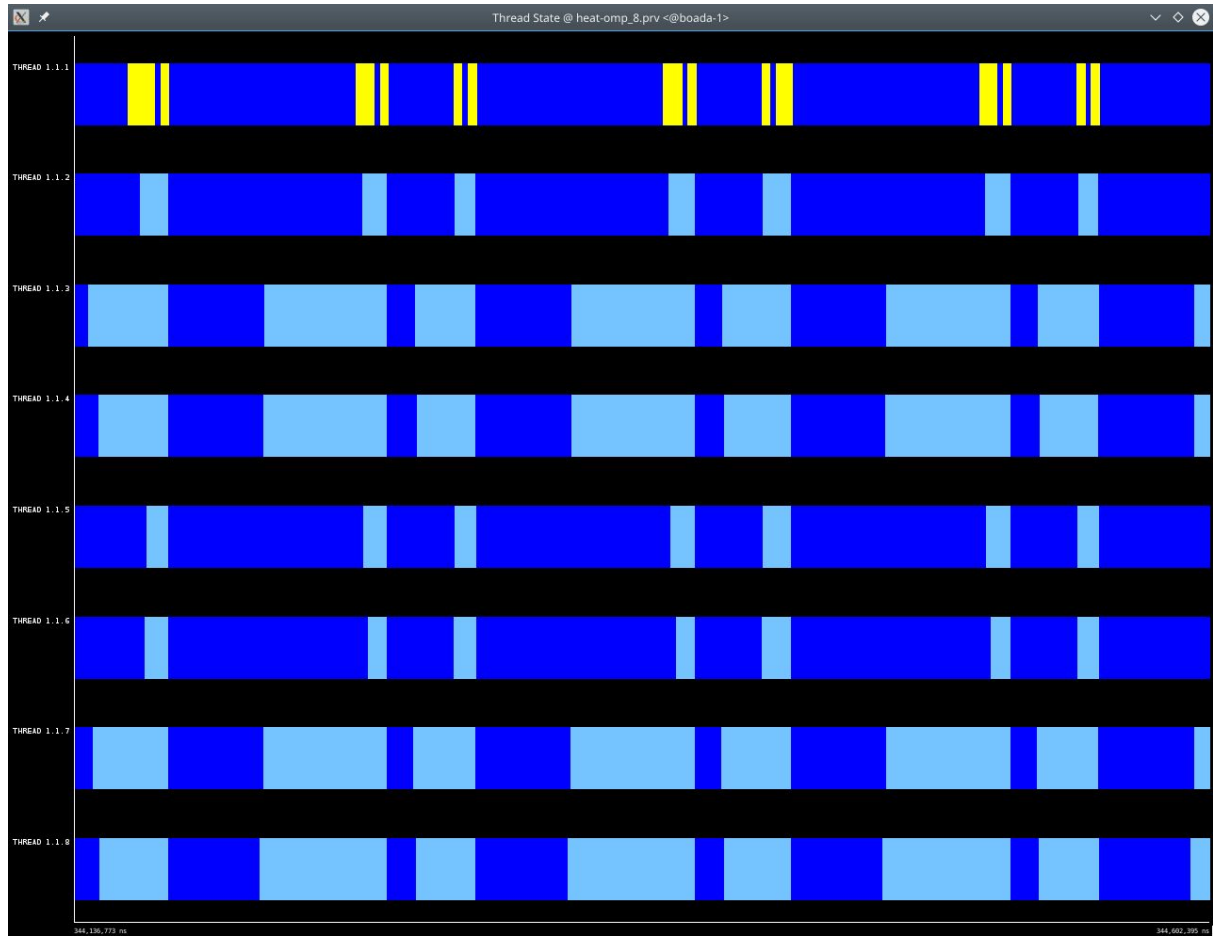


Figure 11: Jacobi trace with 8 threads

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

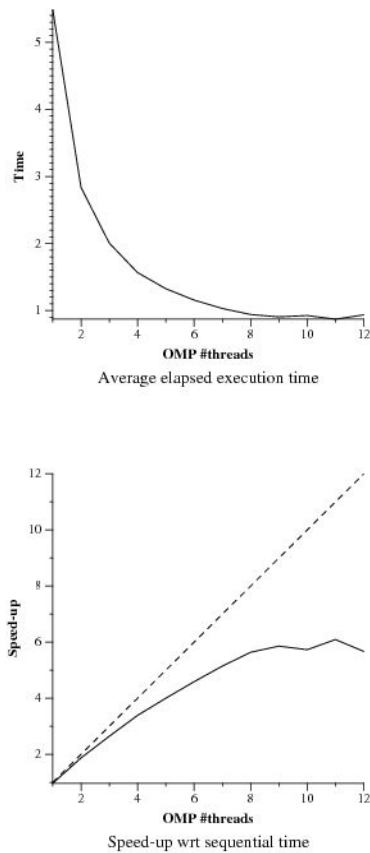


Figure 12: Jacobi plots with 8 threads

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.

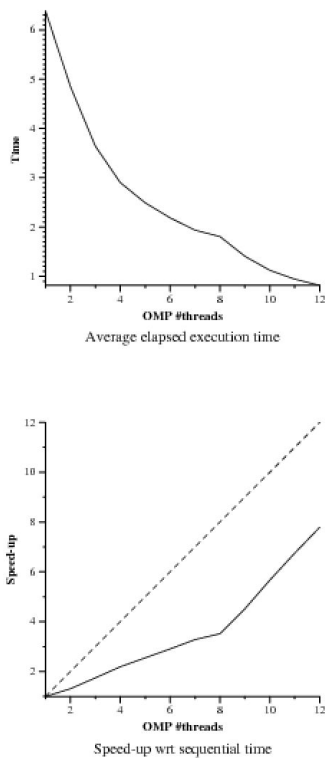


Figure 15: Plots for Gauss version

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