Geometric (data) decomposition: heat diffusion equation

PARALLELISM 2019/2020 Q1

Alejandro Gallego Rodriguez
Oriol Catalán Fernández

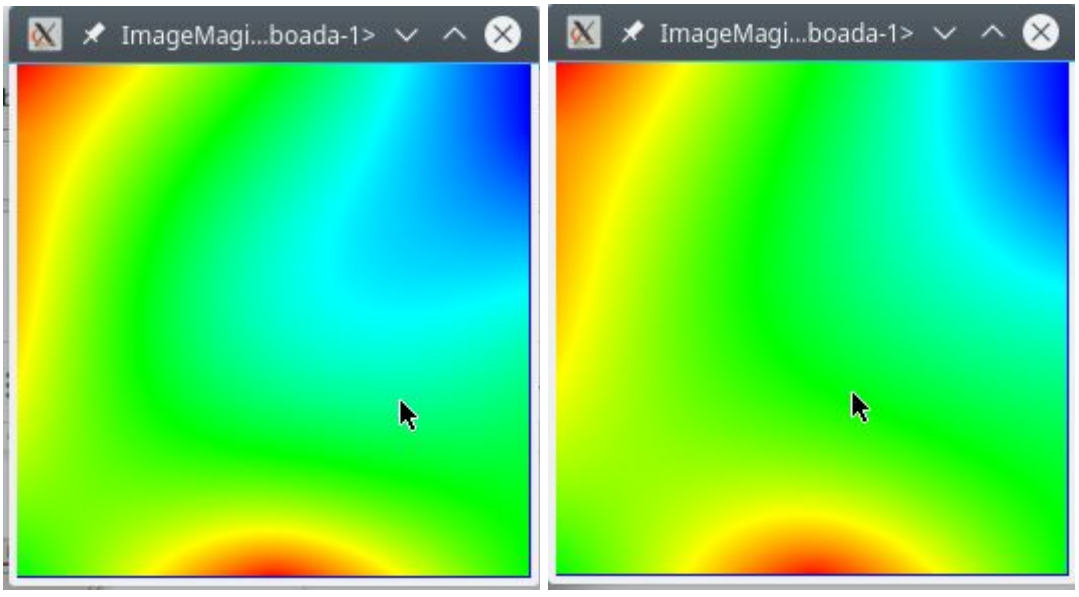
Group 1203
27/12/2019

Introduction

In this assignment we will introduce a new way of analyzing parallelism with the Jacobi and Gauss-Seidel algorithms.

Both algorithms are used to compute the heat given a number of heat sources and a solid, resulting in a 2-dimensional image of the heat zones, shown in the figure below.

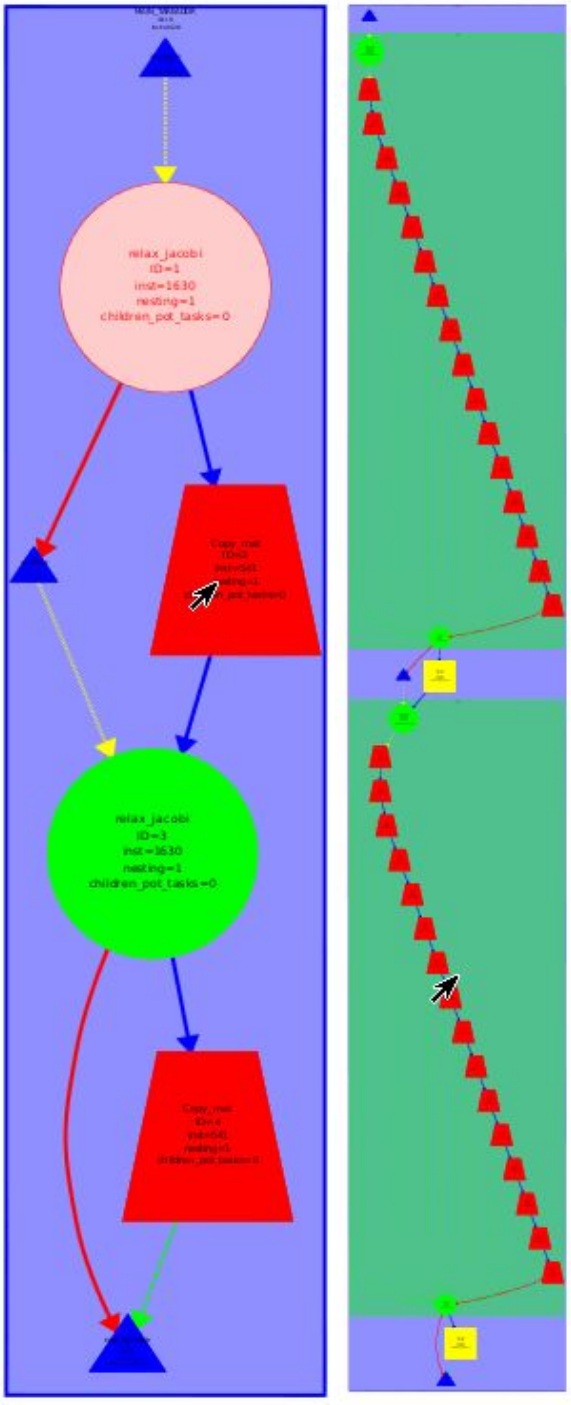
As we see in the images, both algorithms are computed in a different way, thing that results in slightly different images.



Left: Jacobi image, Right: Gauss-Seidel image

Session 1: Analysis of task granularities and dependencies

Jacobi Algorithm



In the first session, we will analyze the dependencies across tasks. For this purpose we will use Tareador, which displays an image representing the tasks and their dependencies.

In the left image we have the original version of the source code, in which a task is declared each time *relax_jacobi* and *copy_mat* are called. We can see that there are 2 tasks each for both functions, this happens because in the tareador version of heat.c, small.dat configuration is used in order to achieve a more simple analysis with 2 iterations. Dependencies are created because *copy_mat* needs the output of *relax_jacobi*, more specific the matrix "*uhelp*", that is computed in *relax_jacobi* and copied into "*u*" in *copy_mat.*

In the right graph, we made a new declaration of task in the innermost loop of the *relax_jacobi* function. This will display the dependencies between the loops inside the function. With Tareador we can see that the "*sum"* variable is dependant on each iteration of the innermost loop. The rest of dependencies previously commented are kept despite our changes in the source code.

Figure 1.1 Jacobi's graphs

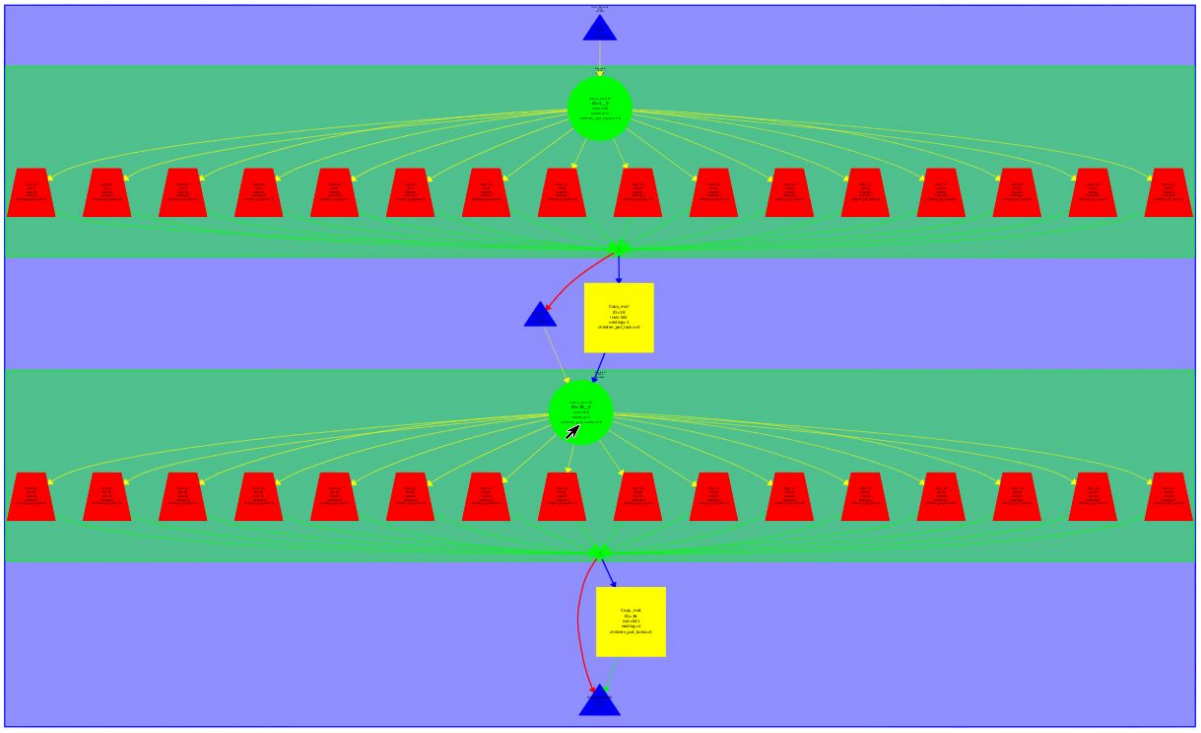
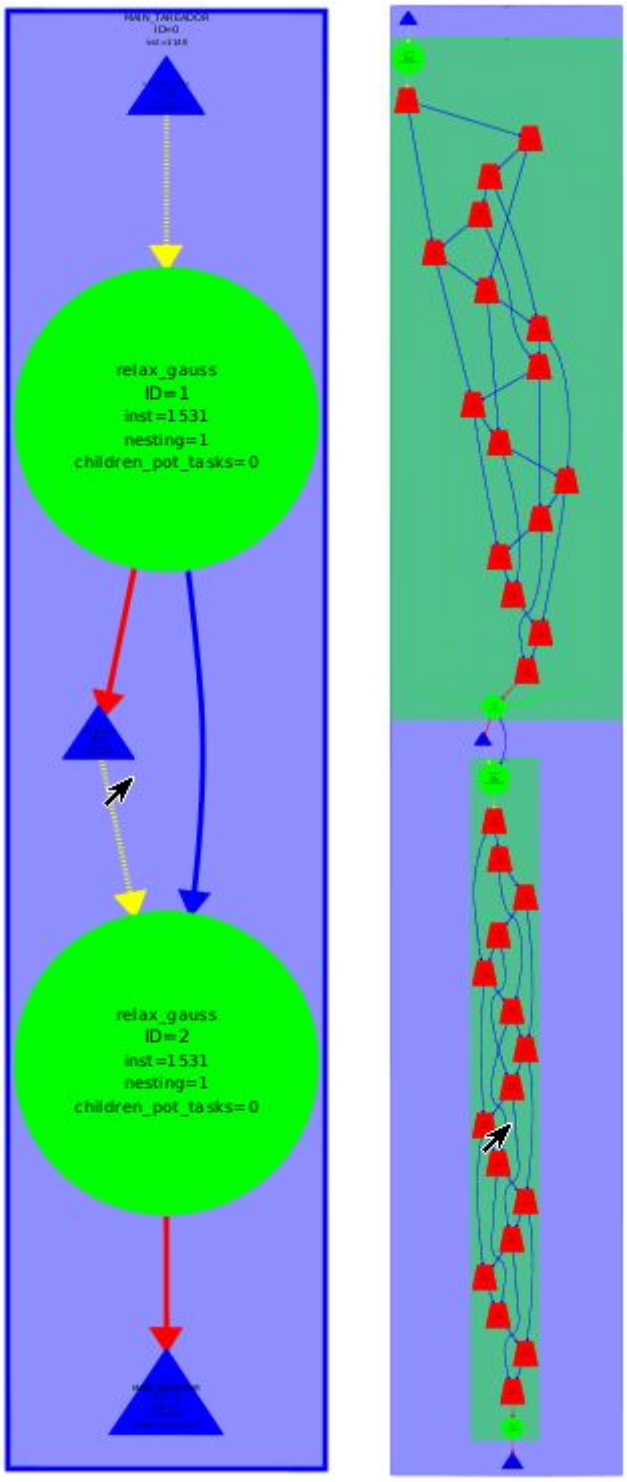


Figure 1.2 Jacobi's graph disabling variable sum

This graph is obtained when we implement the tareador sentence *tareador_disable_object(&sum)* and *tareador_enable_object(&sum)* that disables/enables the object that is passed by parameter, in this case "*sum*", that makes the dependencies of the variable disappear, the consequences is the complete parallelisation of the innermost loop of *jacobi_relax*.

Gauss-Seidel Algorithm



For the Gauss-Seidel algorithm we will repeat the process made for the Jacobi one.
In the left image, the graph given is almost identical to the Jacobi one with the difference that *copy_mat* function is not used. This is because inside the function *relax_gauss* the matrix was directly modified with no need of an auxiliary matrix. The only dependency detected is the loop implicit one.

In the right image, the differences to the Jacobi graph are pretty visual. We can see that each node corresponds to an iteration of the innermost loop. The difference to the other one is that each node has different dependencies. The "*sum*" variable dependency remains from the Jacobi algorithm and is added a new one that appears every 4 nodes. In this algorithm no auxiliar matrix is needed, the computation is implemented in the original matrix by the "*unew*" variable that makes the calculus for the neighbour cells of the cell we are iterating. The dependency will appear when a new cell requires the neighbour cell that has been early modified.

Figure 1.3 Gauss-Seidel's graphs

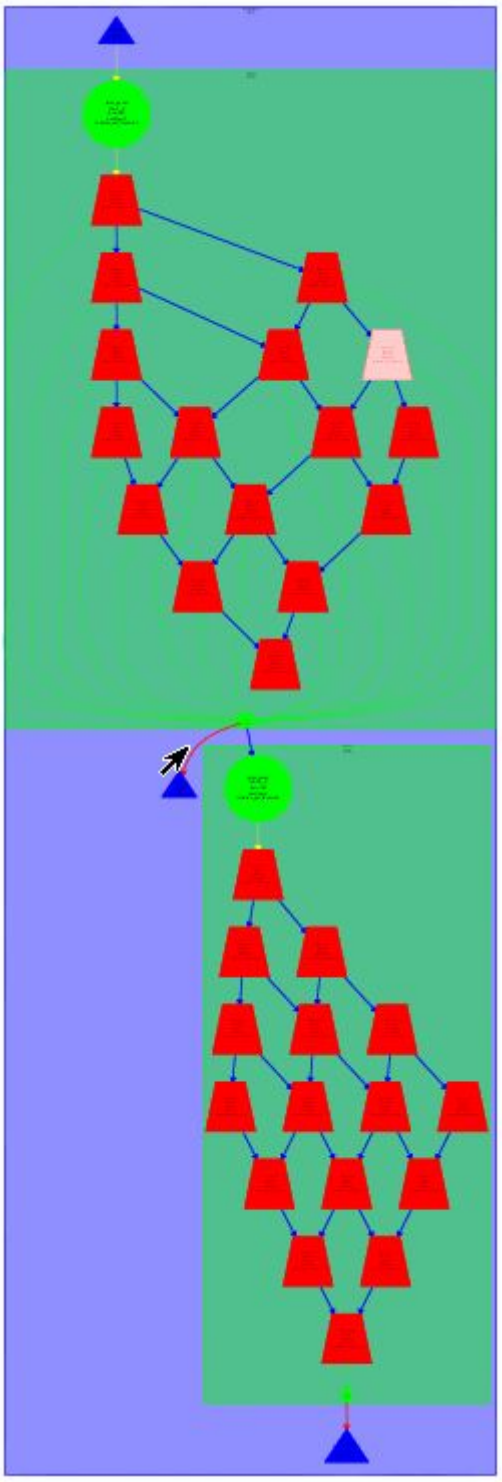


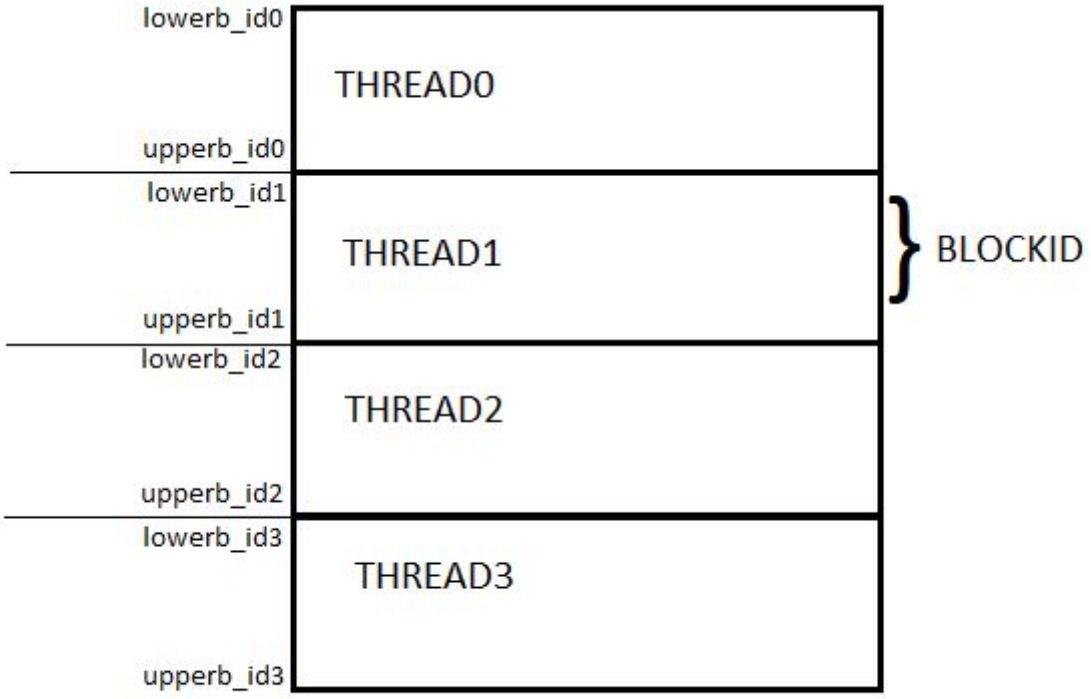
Figure 1.4 Gauss-Seidel's graph disabling variable sum

This graph is obtained when we implement the tareador sentence *tareador_disable_object(&sum)* and *tareador_enable_object(&sum)* that disables/enables the object that is passed by parameter, in this case "*sum*", that makes the dependencies of the variable disappear. The output now only represents the dependencies of the values of i-1 and j-1 cells in the cell identified with i and j.

Session 2: OpenMP parallelization and execution analysis: Jacobi

The Jacobi algorithm, shown below, uses an auxiliary matrix to implement his code. In the utmp matrix we compute the Jacobi heat equation, and then in the main function is copied to the original one.

This is the sketch of geometric data decomposition for howmany = 4:



Our implementation has no need of using the first loop, because we use an intrinsic function to get the thread id and the number of threads.

At first implementing the algorithm we did not make the variable “diff” private, because we thought that due to updating his value in each iteration was not necessary. The code worked fine but was incorrect. The Makefile was done with an “-O3” optimization option that corrected our mistake automatically. When executed with an “-O0” optimization (no optimization) the image displayed was incorrect.

The nature of this error is that the variable “sum” is increased in each iteration by the variable “diff”, what makes the variable “diff” a clear example of data race condition. We solved this by making this variable “private”.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    // int howmany=8;
   // for (int blockid = 0; blockid < howmany; ++blockid) {
      #pragma omp parallel private(diff) reduction(+:sum)
      {
      int my_id = omp_get_thread_num();
      int howmany = omp_get_num_threads();
      int i_start = lowerb(my_id, howmany, sizex);
      int i_end = upperb(my_id, 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 2.1: Source code of Jacobi algorithm with parallelism

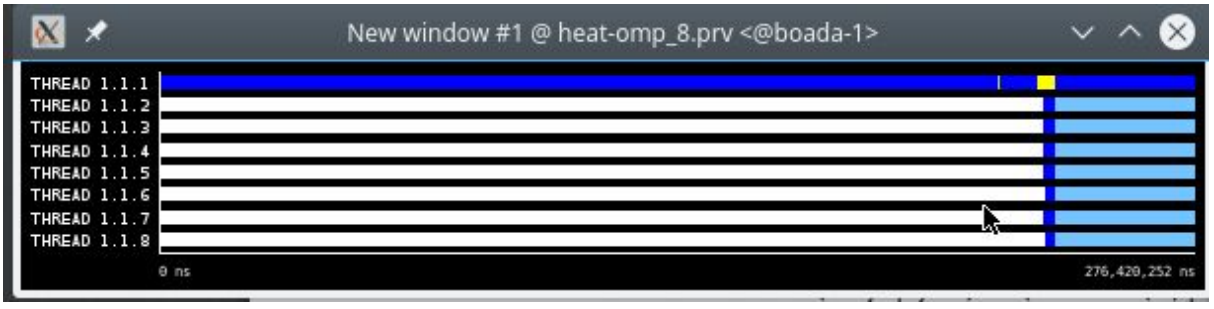


Figure 2.2: Timeline Jacobi solver

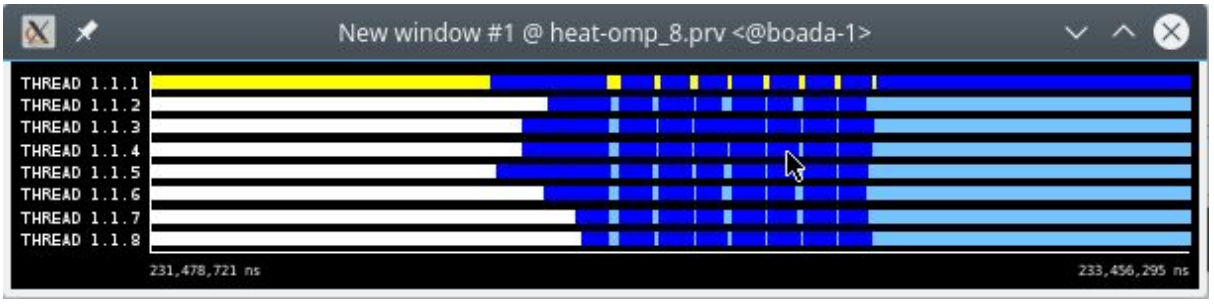
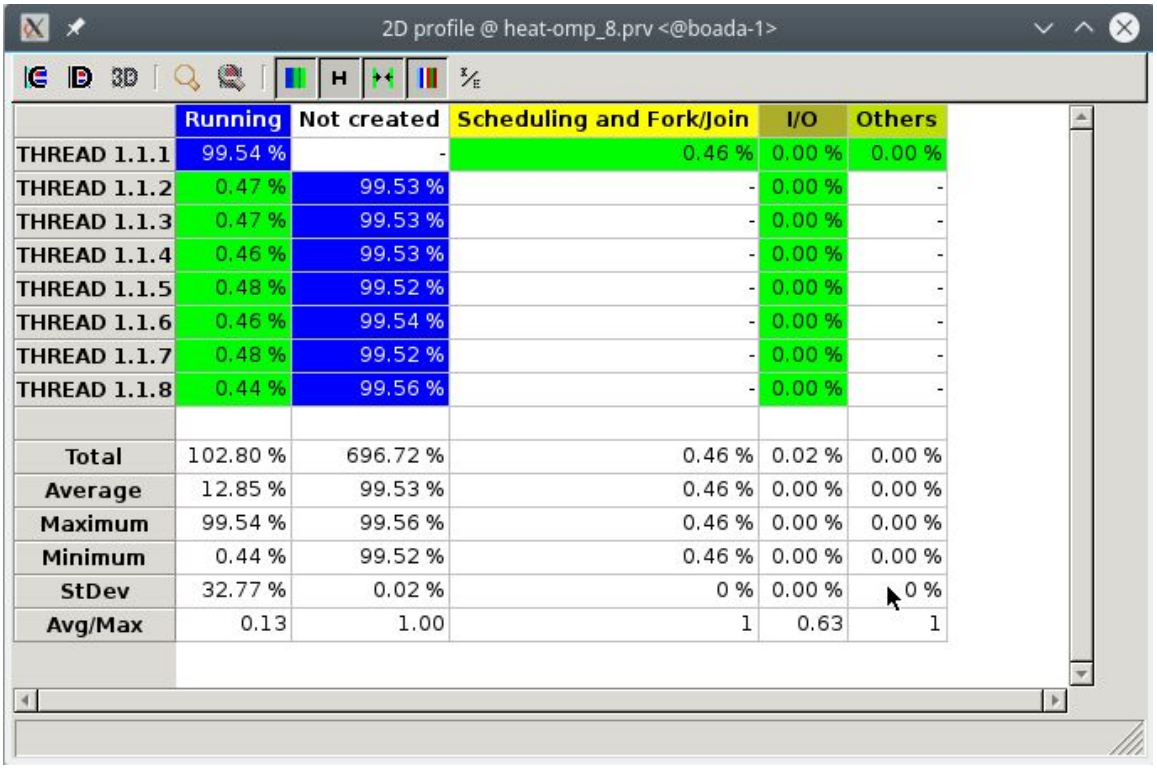


Figure 2.3: Zoomed timeline for Jacobi solver

We can see in the figures above that no synchronisation has been made. This is because we have made use of a data decomposition strategy, dividing the matrix among threads with no need of sharing information between them.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.54 %	-	0.46 %	0.00 %	0.00 %
THREAD 1.1.2	0.47 %	99.53 %	-	0.00 %	-
THREAD 1.1.3	0.47 %	99.53 %	-	0.00 %	-
THREAD 1.1.4	0.46 %	99.53 %	-	0.00 %	-
THREAD 1.1.5	0.48 %	99.52 %	-	0.00 %	-
THREAD 1.1.6	0.46 %	99.54 %	-	0.00 %	-
THREAD 1.1.7	0.48 %	99.52 %	-	0.00 %	-
THREAD 1.1.8	0.44 %	99.56 %	-	0.00 %	-
Total	102.80 %	696.72 %	0.46 %	0.02 %	0.00 %
Average	12.85 %	99.53 %	0.46 %	0.00 %	0.00 %
Maximum	99.54 %	99.56 %	0.46 %	0.00 %	0.00 %
Minimum	0.44 %	99.52 %	0.46 %	0.00 %	0.00 %
StDev	32.77 %	0.02 %	0 %	0.00 %	0 %
Avg/Max	0.13	1.00	1	0.63	1

Figure 2.4: State profile for Jacobi solver

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for 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 ];
}
```

Figure 2.5: Source code of copy_mat function with parallelism

In spite of improving the performance of the algorithm we have parallelized the copy_mat function that takes a great amount of time in its execution.

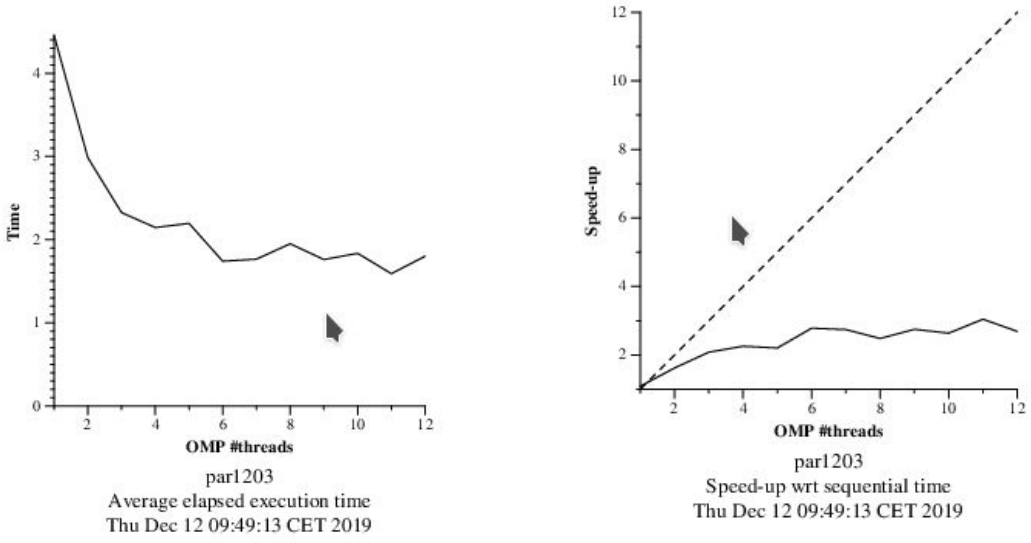


Figure 2.6: Strong scalability plots for Jacobi solver

In the figure above we can see the strong scalability plots for the Jacobi solver. The execution time decreases as long as we increase the number of threads even though there are moments when the time increases a bit.

The speed-up plot is no different from the previous one, it follows the same pattern. There is little improvement with the increasing of threads and also decreasing performance.

Session 3: OpenMP parallelization and execution analysis: Gauss-Seidel

The Gauss-Seidel algorithm follows a more complex mechanism than the Jacobi one. Here we do not need the use of an auxiliary matrix, each iteration the matrix u updates his value u[i][j]. Because of this we need dependency clauses to avoid data race condition.

Each u[i][j] is dependant on u[i-1][j] and u[i][j-1], so we must take this into account at the time of declaring the dependent clauses. In our implementation, we make dependencies among blocks, not iterations as we can see in the code below. Because of this we created a new loop to traverse columns.

In the Jacobi algorithm we make use of a block data decomposition by rows, this means that each thread executes a block of rows. Now in the Gauss-Seidel, we use a block data decomposition as well, but with the difference that the block is a row-column.

As shown in the figure, we made parallelism with blocks and make the dependencies with the blocks themselves. With the “sink” clause in the depend construct we set a waitpoint to pause the execution until the indexes inside are executed. The “source” clause determines when the dependencies declared before are completed.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    #pragma omp parallel private(diff, unew) reduction(+:sum)
    {
      int P = omp_get_num_threads();
      int howmanyi = P;
      int howmanyj = P;
      #pragma omp for ordered(2) schedule(static)
      for (int blockidi = 0; blockidi < howmanyi; ++blockidi) {
        for (int blockidj = 0; blockidj < howmanyj; ++blockidj) {
          int i_start = lowerb(blockidi, howmanyi, sizex);
          int i_end = upperb(blockidi, howmanyi, sizex);
          int j_start = lowerb(blockidj, howmanyj, sizey);
          int j_end = upperb(blockidj, howmanyj, sizey);
           #pragma omp ordered depend(sink: blockidi-1, blockidj) depend(sink: blockidi, blockidj-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;
}
                                                                                79,1
```

Figure 3.1: Source Code of Gauss-Seidel algorithm with parallelism

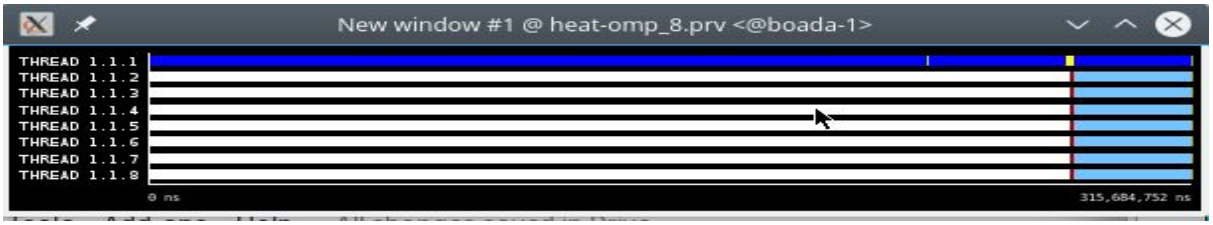


Figure 3.2: Timeline of the Gauss-Seidel solver

This algorithm follows the same logic as the Jacobi one, the major part of the execution resides in a sequential part that is major compared to the parallelised one

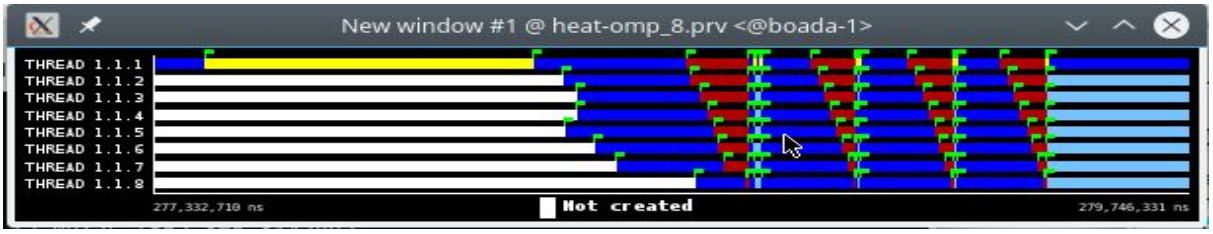
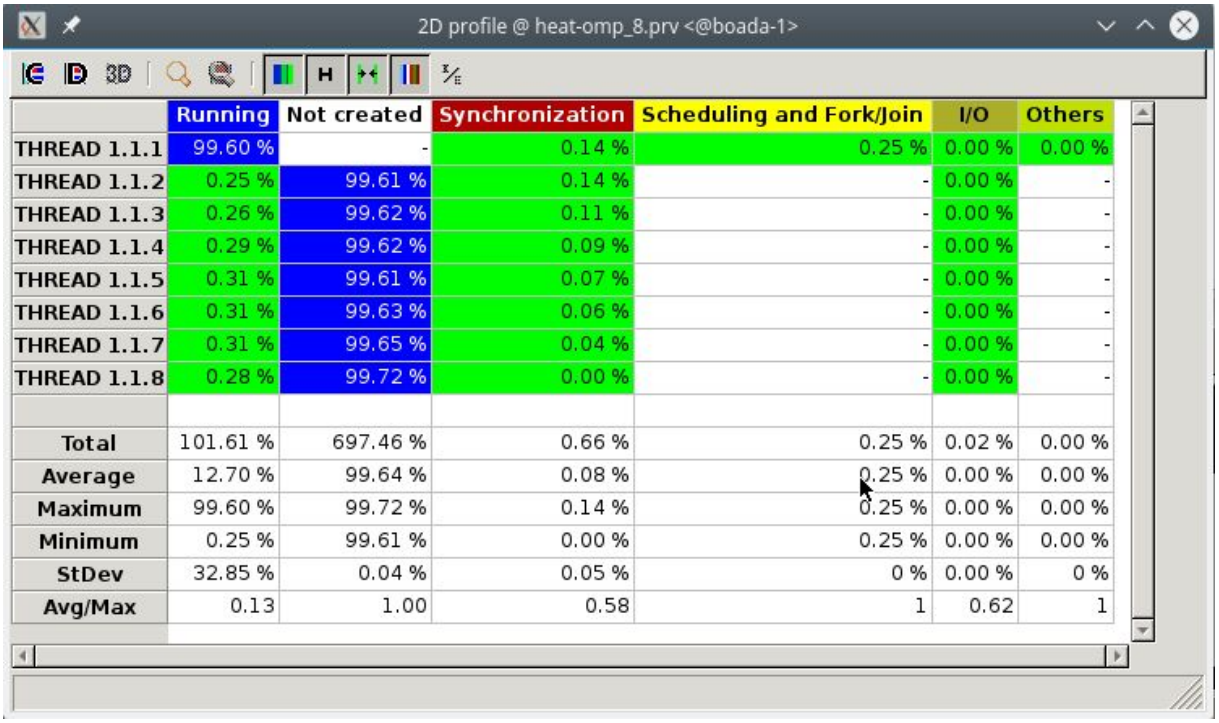


Figure 3.3: Zoomed timeline of the Gauss-Seidel solver

Following the previous paragraph, the image above is the zoomed part of the parallelised part of the Figure 3.2. As we can see, the parallel region begins with Thread 1.1.1. forking the threads that are going to be used. Then the threads run in parallel with some synchronisation parts that are determined by the number of iterations



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.60 %	-	0.14 %	0.25 %	0.00 %	0.00 %
THREAD 1.1.2	0.25 %	99.61 %	0.14 %	-	0.00 %	-
THREAD 1.1.3	0.26 %	99.62 %	0.11 %	-	0.00 %	-
THREAD 1.1.4	0.29 %	99.62 %	0.09 %	-	0.00 %	-
THREAD 1.1.5	0.31 %	99.61 %	0.07 %	-	0.00 %	-
THREAD 1.1.6	0.31 %	99.63 %	0.06 %	-	0.00 %	-
THREAD 1.1.7	0.31 %	99.65 %	0.04 %	-	0.00 %	-
THREAD 1.1.8	0.28 %	99.72 %	0.00 %	-	0.00 %	-
Total	101.61 %	697.46 %	0.66 %	0.25 %	0.02 %	0.00 %
Average	12.70 %	99.64 %	0.08 %	0.25 %	0.00 %	0.00 %
Maximum	99.60 %	99.72 %	0.14 %	0.25 %	0.00 %	0.00 %
Minimum	0.25 %	99.61 %	0.00 %	0.25 %	0.00 %	0.00 %
StDev	32.85 %	0.04 %	0.05 %	0 %	0.00 %	0 %
Avg/Max	0.13	1.00	0.58	1	0.62	1

Figure 3.4: State Profile of the Gauss-Seidel solver

Figure 3.4 gives us a more specific information about the time spent by threads in synchronisation, scheduling, forking, running, etc.

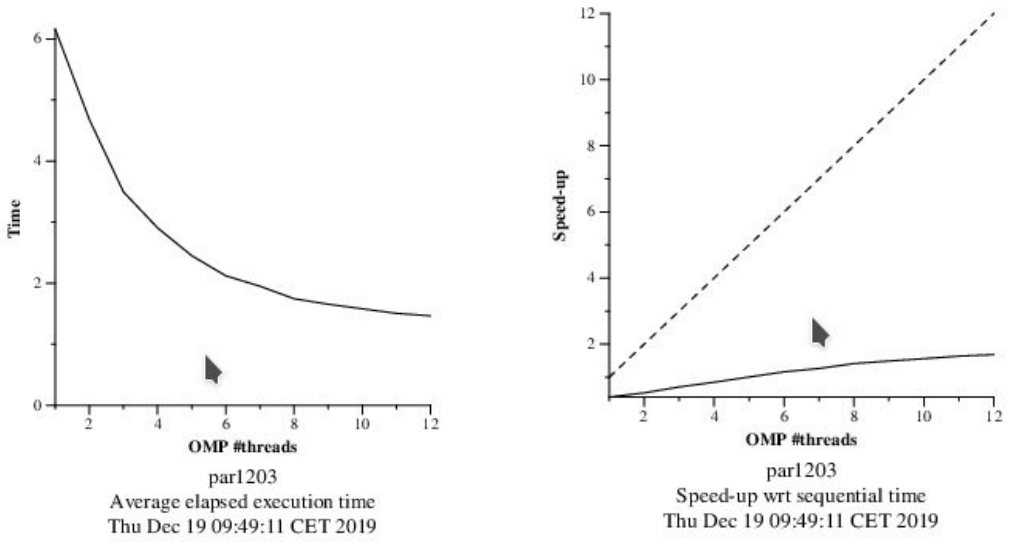


Figure 3.5: Strong scalability plots for Gauss-Seidel algorithm

The strong scalability plots show a mediocre, but a more constant, improvement compared to the Jacobi algorithm.

The left plot shows us a good performance among the execution time with more threads.

On the other hand, the right plot gives us very little improvement, but we must see that with 1 thread the speed-up does not begin in 0, but below that (we can see this in the ideal speed-up line), with this we can determine that even though the line is far away from the ideal one, improvement is made.

Finally for 8 threads, we divide the matrix into 8 pieces and have each thread to execute one. This value is obtained at computing the diagonal, that it is the longest block to execute and get its value.

Conclusions

To sum up our last assignment, with the picked up data we can determine that the Jacobi algorithm is slightly more parallelizable than the Gauss-Seidel one, as we saw in both strong scalability plots. This does not surprise us because in the first session with the Tareador application we saw both parallelized dependency graphs and the Jacobi one was far better, because of the matrix dependencies.

Despite this statement, the execution time of the Gauss-Seidel solver does surpass the Jacobi one. This happens because the Gauss-Seidel algorithm does not need a “copy_mat” function due to the lack of auxiliary matrix and the ongoing update of the original matrix as the iterations are executed.

At the time of deciding the best algorithm, we had some difficulties. We finally decide that if application is meant to exploit parallelism and has a sufficient amount of threads , Jacobi algorithm is best, because the Gauss-Seidel solver does not take profit of this advantage and the threads remain idle a considerable amount of time.

On the other hand, if your goal is the execution time, although the difference between them is not large, Gauss-Seidel solver is the better choice despite the waste of resources.