LAB 5 DELIVERABLE
PAR 2108

Raül Montoya Pérez
Miguel Ángel Álvarez Vázquez

Index

Introduction

In this deliverable, we are going to study different ways to parallelise a code that simulates heat diffusion in a solid body using two different solvers for the heat equation, *Jacobi* and *Gauss-Seidel*.

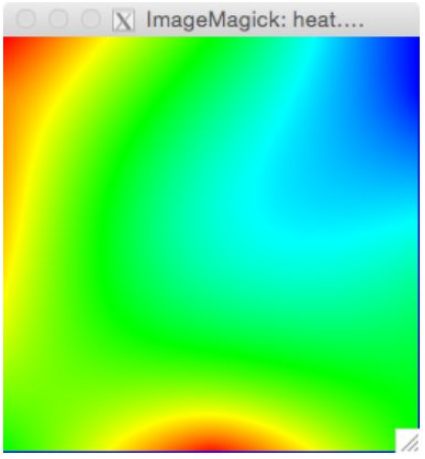
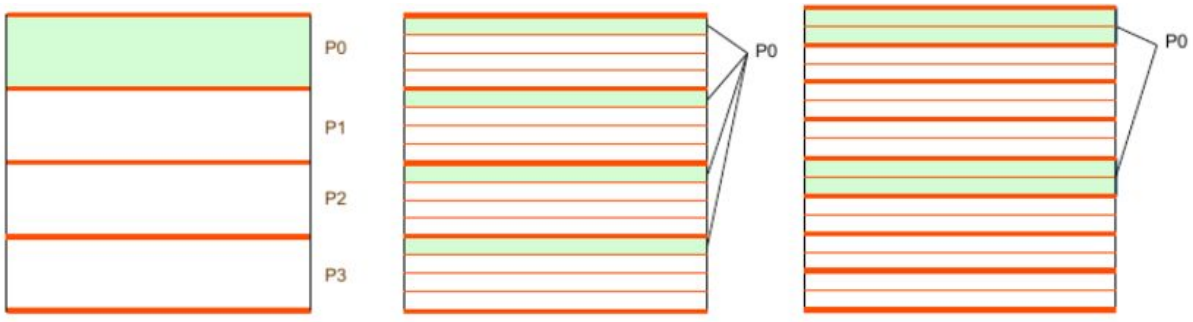


Image generated by heat

In this laboratory sessions, wi will also take a look at different parallelization with data decomposition. Data decomposition is used to derive concurrency problems that operate on large amount on data giving each task less data to be used and based on the multiplicity of the data.

To use the data decomposition correctly, we must take into account the sizes of input or output data, then we can use this structure to decompose the data. Then the structure is divided between tasks, following one of these data distributions: Block distribution, Cyclic distribution or Block-Cyclic distribution.



Block distribution - Cyclic distribution - Block-cyclic distribution

The *Jacobi* algorithm uses only one matrix and do the operations in the same matrix so the result will be affected for each partial operation done. On the other hand, the *Gauss-Seidel*

algorithm uses an auxiliary matrix to save the partial operations, so all the operations are done from the initial values of the matrix.

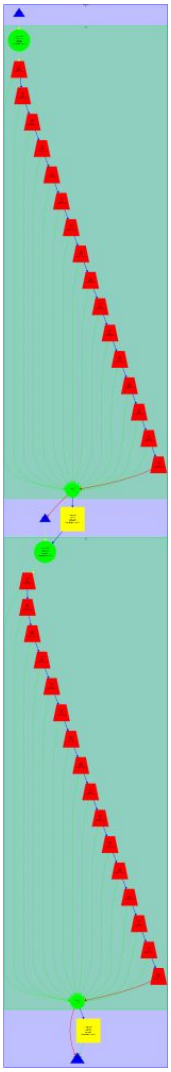
Analysis with Tareador

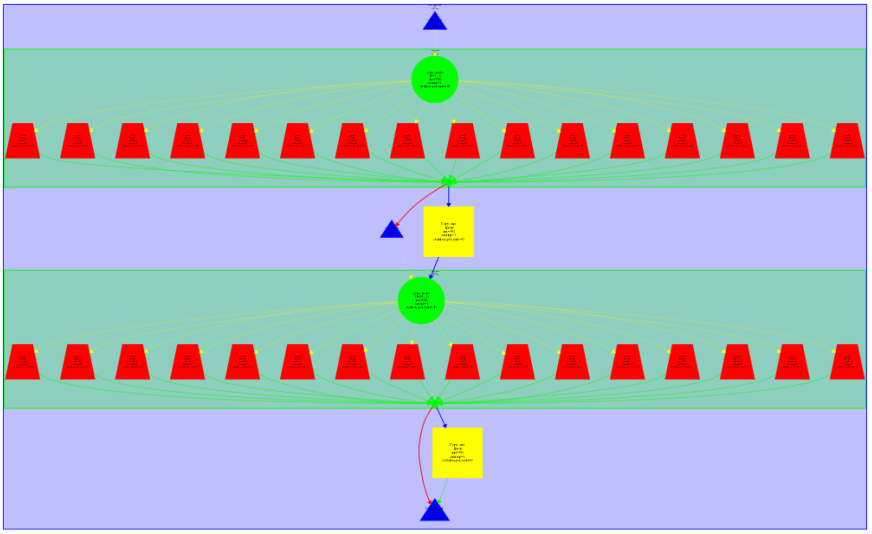
1. Include the relevant parts of the modified solver-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear in the two solvers: Jacobi and Gauss-Seidel. How will you protect them in the parallel OpenMP code?

Edited code Jacobi:

```
for (int j=1; j<= sizey-2; j++) {
        tareador_start_task("jacobi");
        // With this line, we tell Tareador where the task begins
        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);
        // With this disable and enable object line, we tell Tareador to erase the
                dependency created with the variable, in this case, sum
        sum += diff * diff;
        tareador_enable_object(&sum);
        tareador_end_task("jacobi");
        //Here, we tell Tareador where the task ends
    }
```

With this two images, we can see executions for the Jacobi code, with and without disabling the variable “sum”, as we can observe in the first, there exists a huge depence on the “sum” variable. On the second image, we have the execution of heat, but disabling sum so we can achieve a better parallelization.

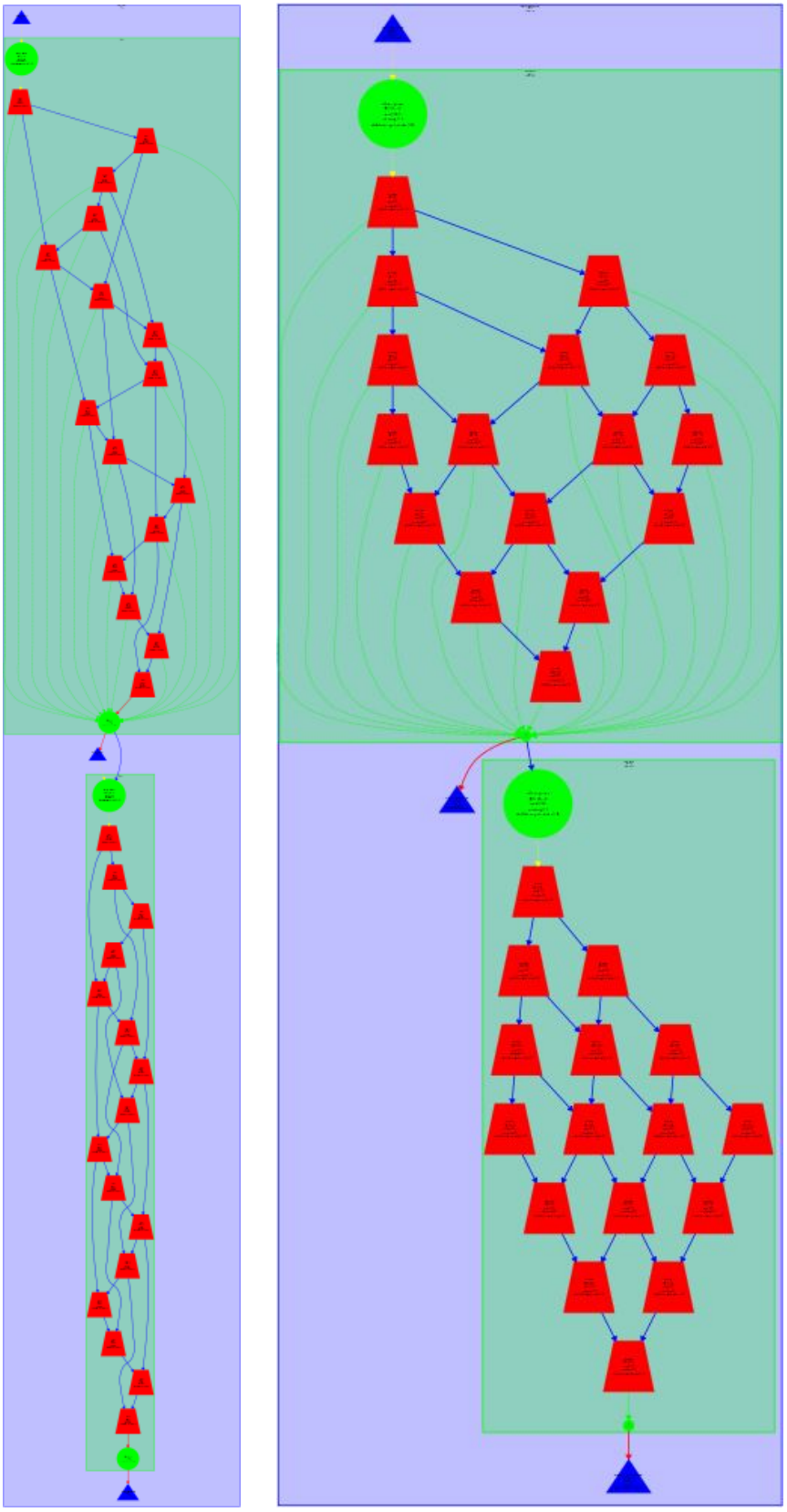




Edited code Gauss-Sielder:

```
for (int j=1; j<= sizey-2; j++) {
        tareador_start_task("gauss");
        // With this line, we tell Tareador where the task begins
        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);
        // With this disable and enable object line, we tell Tareador to erase the
                dependency created with the variable, in this case, sum
        sum += diff * diff;
        tareador_enable_object(&sum);
        u[i*sizey+j]=unew;
        tareador_end_task("gauss");
        //Here, we tell Tareador where the task ends
   }
```

For the Gauss parallelization we followed the same strategy as before. The first image shows the execution without disabling “sum” and the second one disabling it.



OpenMP parallelization and execution analysis: *Jacobi*

1. Describe the data decomposition strategy that is applied to solve the problem, including a picture with the part of the data structure that is assigned to each processor.

We use a geometric block data decomposition, where a block which is assigned to each one thread, is composed by size/howmany rows. So Thread 0 will execute the first size/howmany rows, Thread 1 the next size/howmany rows and so on with the other threads.

Thread 0
Thread 1
Thread 2
Thread 3

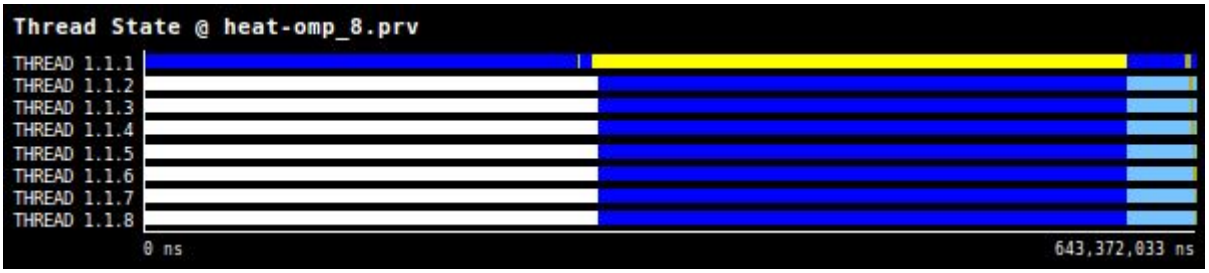
2. Include the relevant portions of the parallel code that you implemented to solve the heat equation using the Jacobi solver, commenting whatever necessary. Including captures of Paraver windows to justify your explanations and the differences observed in the execution.

```
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 private(diff) reduction(+:sum)
//We added this pragma in order to each thread has a private sum variable, and when all threads finish their work, the sum of the private sum variables will be stored in the sum variable. Also, with the clause private(diff) each thread will have a local copy of that variable.
        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;
                        }
                }
```

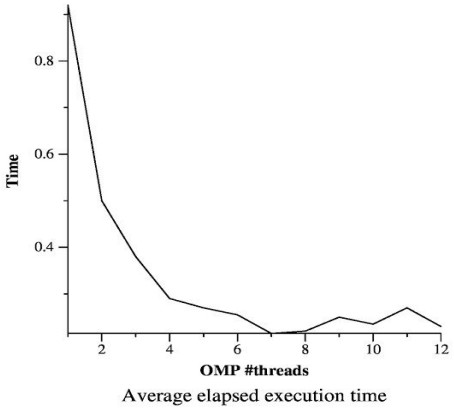
```
        }
        return sum;
}
```

We also parallelized the function copy_mat with #pragma omp parallel for in order to improve the parallelism.

Here is the Paraver execution trace generated:



3. Include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed.

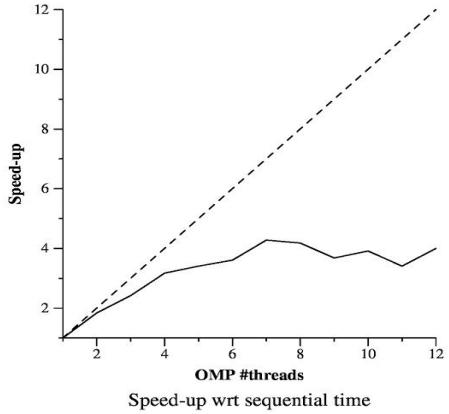


Average elapsed execution time

In the plots we can see that the execution time with the Jacobi implementation decreases while the number of threads executing the code is increased.

As consequence, the speed-up increases when the number of threads is also increased.

However, when the seventh thread is reached, the speed-up changes due to synchronization problems.



Speed-up wrt sequential time

OpenMP parallelization and execution analysis: *Gauss-Siedel*

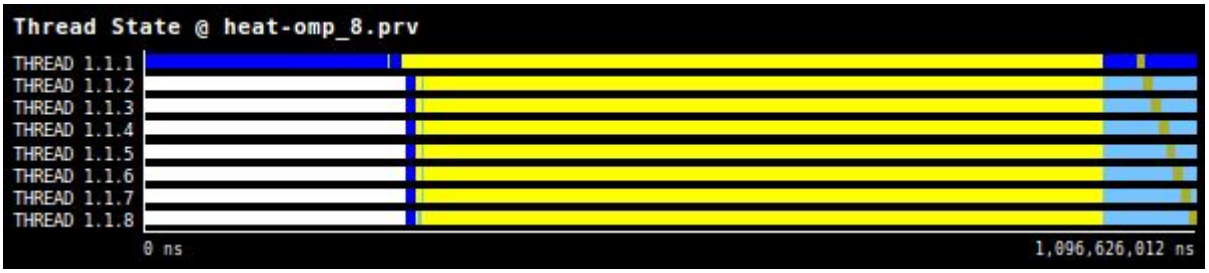
1. Include the relevant portions of the parallel code that implements the Gauss-Seidel solver, commenting how you implemented the synchronization between threads.

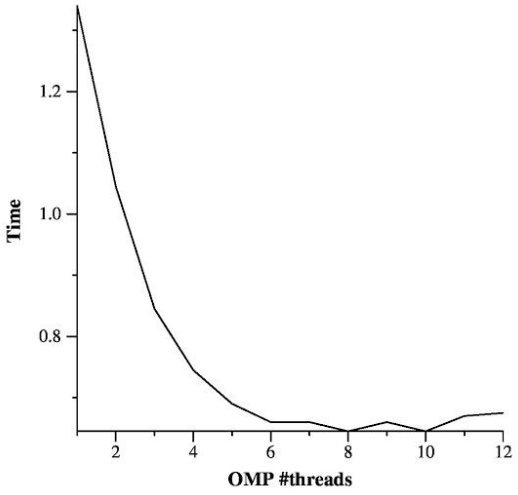
```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
   double unew, diff, sum=0.0;
   int howmany=omp_get_max_threads();
   int howmanyc = 4;

   #pragma omp parallel for schedule(static) ordered (2) private(diff) reduction(+:sum)
//We added this pragma in order to each thread has a private sum variable, and when all threads finish their work, the sum of the private sum variables will be stored in the sum variable. Also, with the clause private(diff) each thread will have a local copy of that variable. With the clause ordered(2) we say that in the next 2 loops we use a ordered depend pragma clause.
   for (int blockidf = 0; blockidf < howmany; ++blockidf) {
      for(int blockidc = 0; blockidc < howmanyc; ++blockidc) {
         int j_start = lowerb(blockidc, howmanyc, sizey);
         int j_end = upperb(blockidc, howmanyc, sizey);
         int i_start = lowerb(blockidf, howmany, sizex);
         int i_end = upperb(blockidf, howmany, sizex);
        #pragma omp ordered depend(sink: blockidf-1, blockidc) depend(sink: blockidf, blockidc-1)
//We added this pragma in order to indicate the dependencies between each position of the matrix. Each position has a dependency with the element on the left and with the element above.
         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)
//We added this pragma in order to indicate that the previous ordered depend pragma ends here
      }
   }
   return sum;
}
```

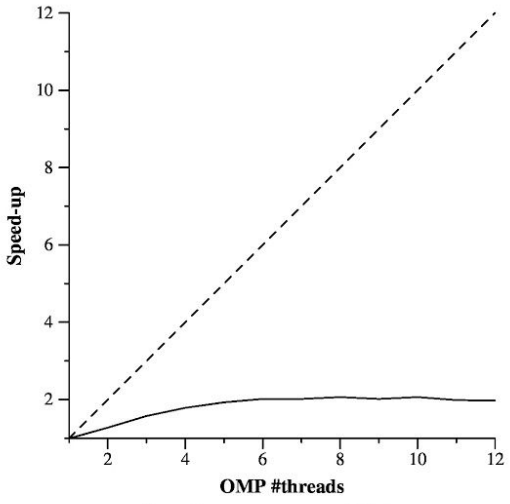
2. Include the speed–up (strong scalability) plot that has been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

Here is the Paraver execution trace generated:





Average elapsed execution time



Speed-up wrt sequential time

In the plots we can see that the execution time with the Gauss implementation decreases while the number of threads executing the code is increased.

As consequence, the speed-up increases when the number of threads is also increased.

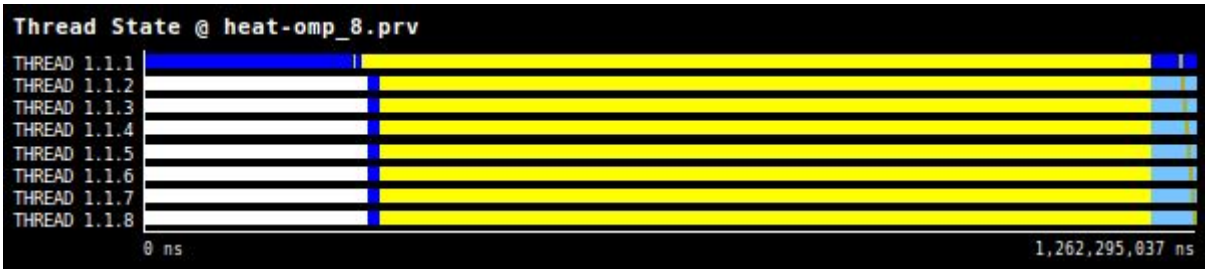
However, when the sixth thread is reached, the speed-up doesn't increase more.

Finally, if we compare the plots of Jacobi and Gauss implementation, we see that the second one is worse than the first due to the matrix dependencies.

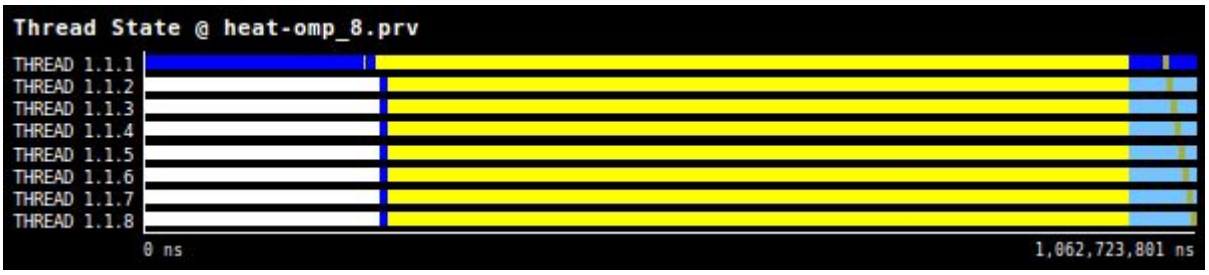
3. Explain how did you obtain the optimum value for the ratio computation/synchronization in the parallelization of this solver for 8 threads

We executed the program with 2500 iterations and resolution of 254 for 2, 4, 8, 16 and 32 blocks and we have got the next results:

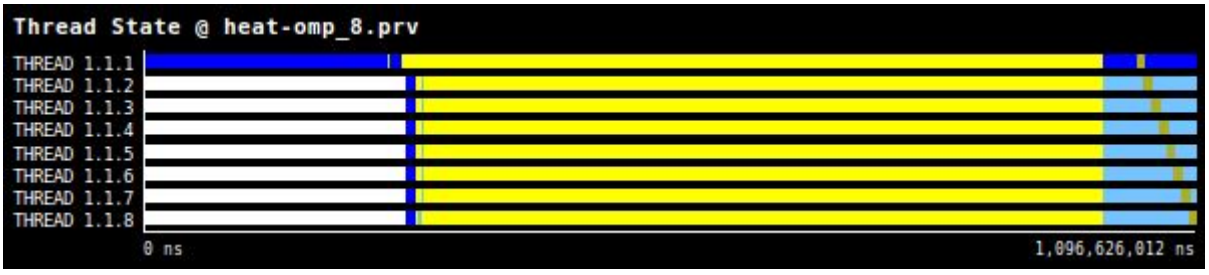
2 blocks:



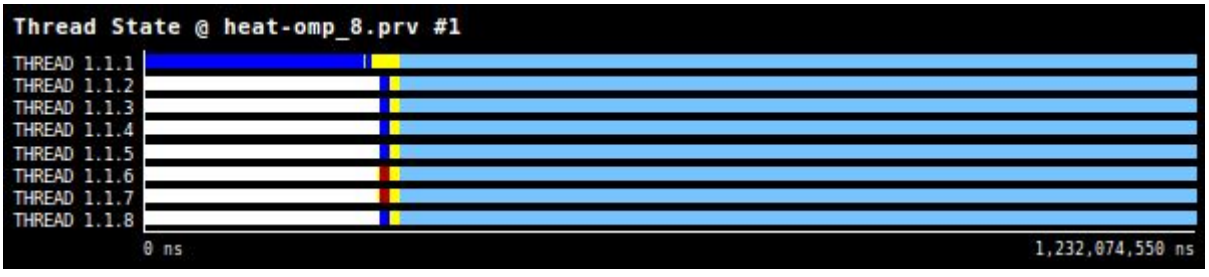
4 blocks:



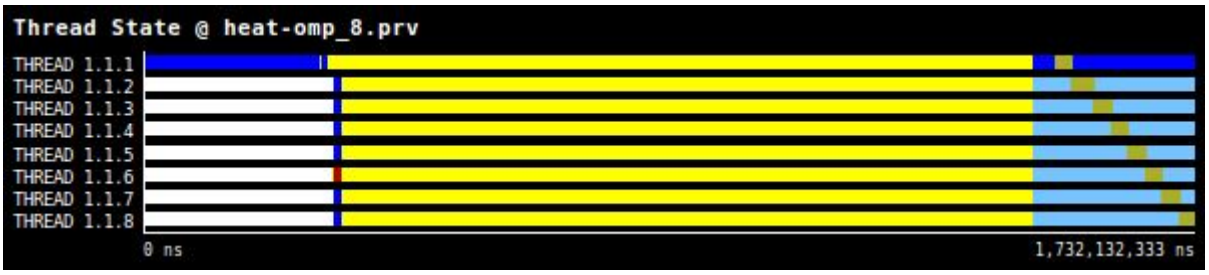
8 blocks:



16 blocks:



32 blocks:



As we can see, the best execution time obtained is with 4 blocks. That's because with less blocks there is a lot of computation per thread, and with more blocks the time wasted on synchronizations is greater.

Optional

1. Implement an alternative parallel version for Gauss-Seidel using #pragma omp task and task dependences to ensure their correct execution. Compare the performance against the #pragma omp for version and reason about the better or worse scalability observed.

We think that the best option is to create a task for each block with #pragma omp task clause with an input dependences of the left block and the block above, and output dependences with the address of the block to compute.

In order to fix the false sharing that it will be produced with the variable sum, we can create an array with as many lines as threads we have available and each of the lines will have a size equal to the size of the line of the cache:
Double local_sum[num_threads][ElementsPerLine]

So each thread will increase its value in the first position of each line:
Local_sum[ID_Thread][0] += diff*diff;

When all the tasks finish their work, all the local sum will be added and saved into sum.

Conclusions

The execution of a program can be affected by the processors coherence systems, as we saw in the previous exercises. Due to this, we conclude that it is an important part of the program that has to be taken into account. Also, we can say that in some cases a data decomposition strategy improves a task decomposition strategy, for example, in the Gauss program.