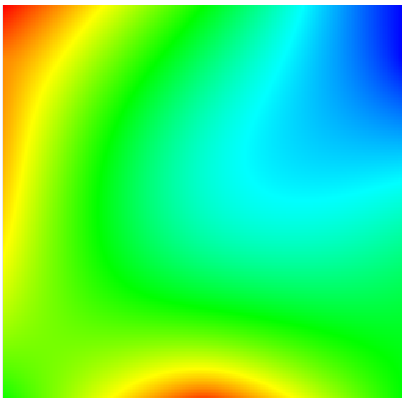
PAR Laboratory Assignment

Lab 5: Geometric (data) decomposition: solving the heat equation

PAR 2312
Fèlix Arribas Pardo
Ana Mestre Borges

1. Introduction

In this laboratory session we are going to practice and observe different parallelization strategies on a sequential code: heat.c. This code simulates heat diffusion in a solid body sing two different solvers for the heat equation (Jacobi and Gauss-Seidel)



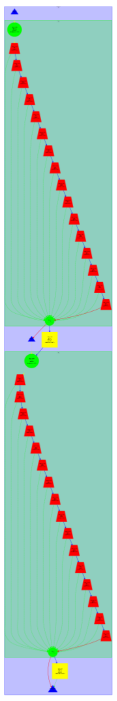
First we'll study the behaviour for both codes given: the Jacobi and the Gauss-Seidel and then achieve an improved implementation for each of them.

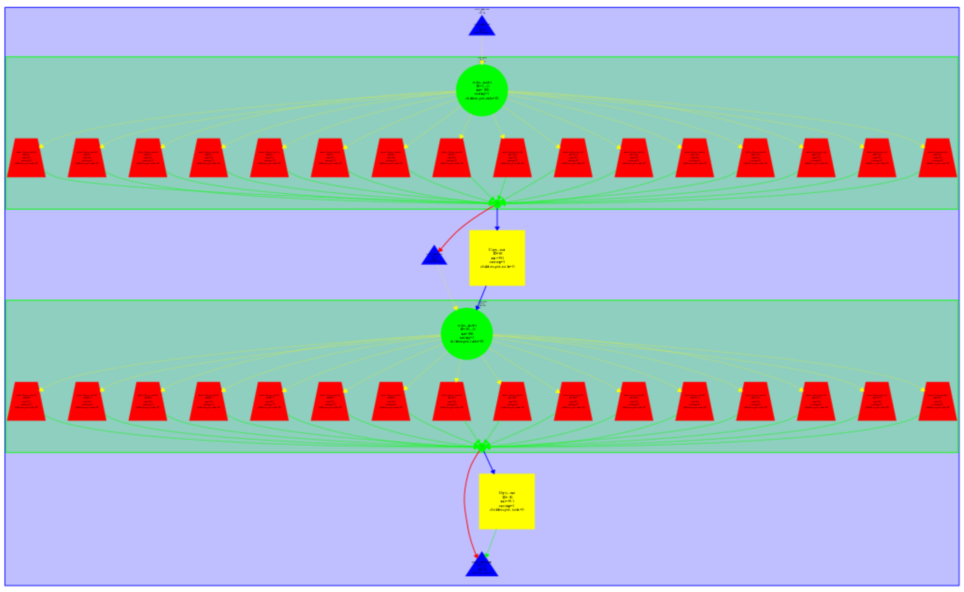
Jacobi

As we were understanding the code we noticed that the sum variable caused dependency and we disabled it and also we created taks with tareador in the most inner loop.

```
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_loop_jacobi"); // TAREADOR
            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"); // TAREADOR
        }
      }
    }
    return sum;
}
```

On these first two pictures it is obviously seen the dependence over the sum variable by changing its behaviour. On the one hand we have the tasks generated without disabling the sum variable and on the other hand we have the same but now disabling it.



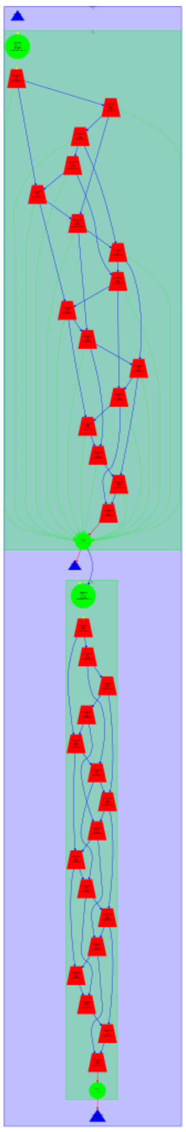


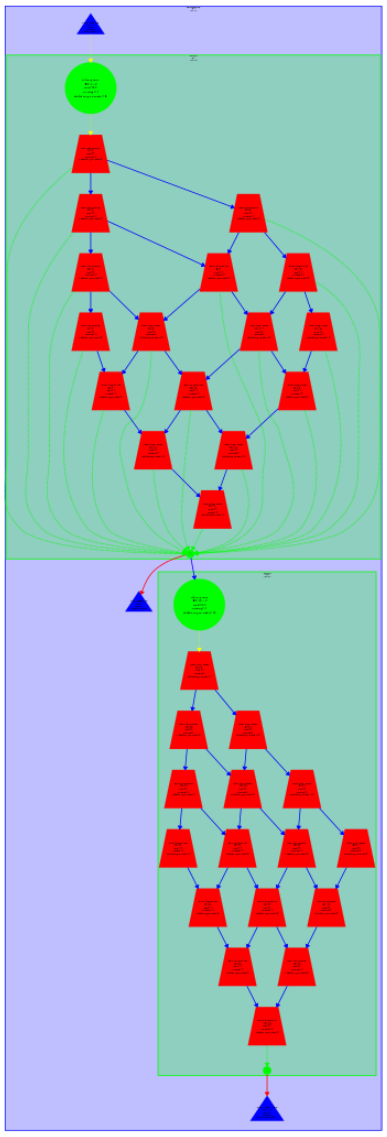
Gauss - Seidel

```
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_loop_gauss"); // TAREADOR
          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"); // TAREADOR
```

```
            }
          }
        }
        return sum;
    }
```

With the Gauss-Seidel code we've followed the exact same strategy, adding a tareador task to the inner loop and disabling and enabling again the sum variable. The changes are clearly showed in the following tareador pictures. First without disabling sum and second with the improvement of disabling it.





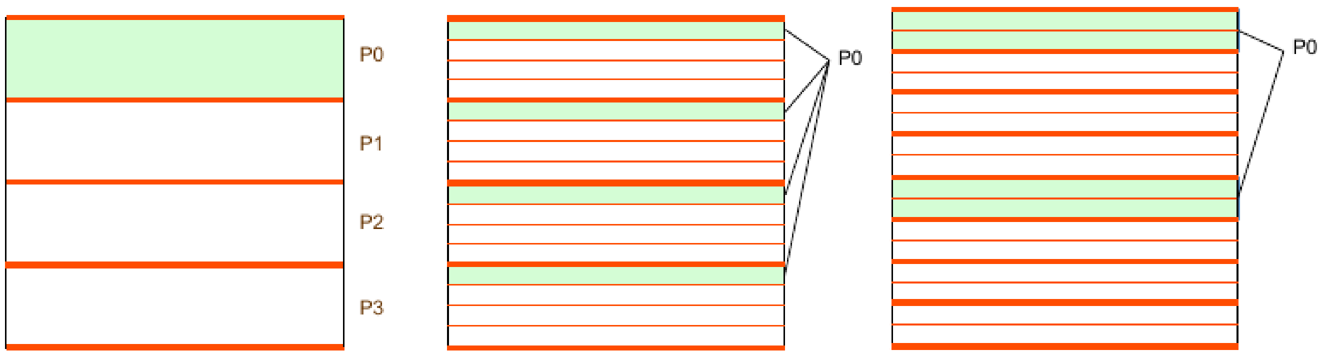
2. Parallelization strategies

Parallelization with data decomposition.

In this session we are going to parallelize the Jacobi and Gauss-Seidel codes using **#pragma omp for** and applying a geometric data decomposition strategy.

Data decomposition is used to derive concurrency for problems that operate on large amounts of data focusing on the multiplicity of data.

How to apply data decomposition? First either the input or output data structures have to be chosen in order to use them as a structure for the decomposition. Then this structure is partitioned across tasks following a data distribution like the following: block, cyclic or block-cyclic.

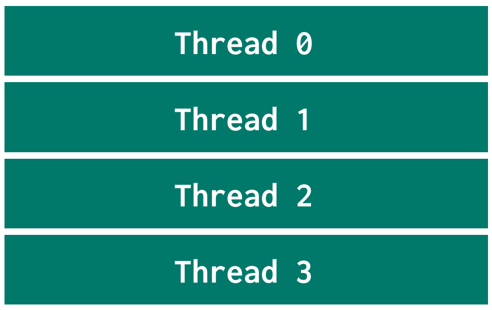


Block distribution - Cyclic distribution - Block-cyclic distribution

3. Performance evaluation

Parallelization of Jacobi with OpenMP for

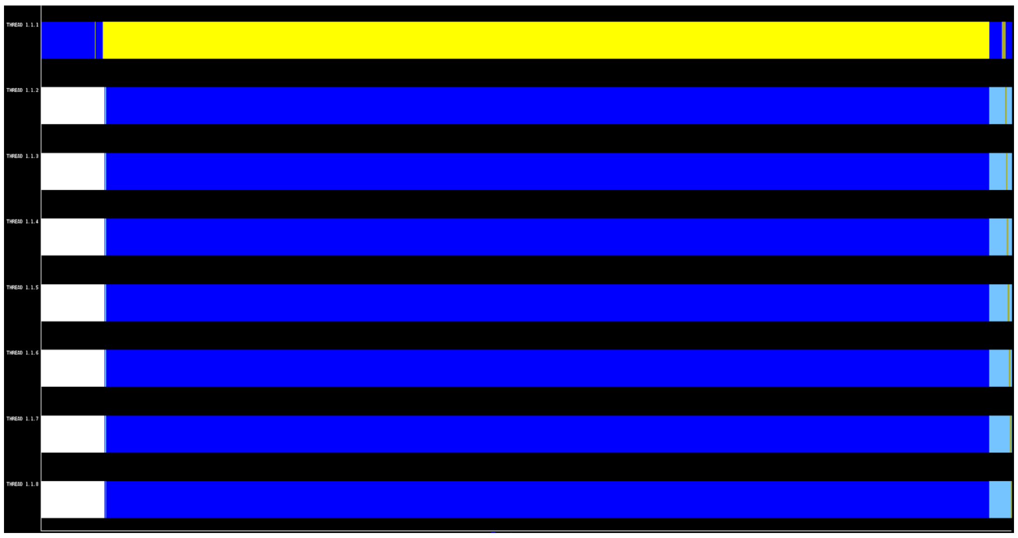
The data decomposition strategy used for this section is the following. We have a table of size N, then we divide N by the number of threads and the result obtained is this one:



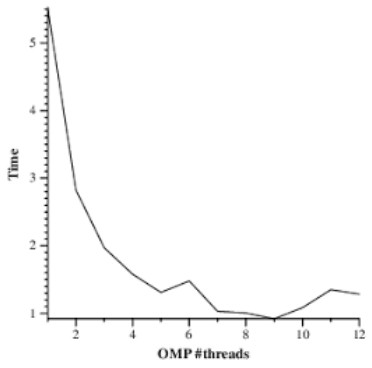
Here's the code used for the Jacobi implementation. We used **`#pragma omp parallel for`** in the first loop, making the diff variable privative and using a reduction for the sum variable as we wanted to protect the value of it from the dependence generated in the code.

```
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)
    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;
}
```

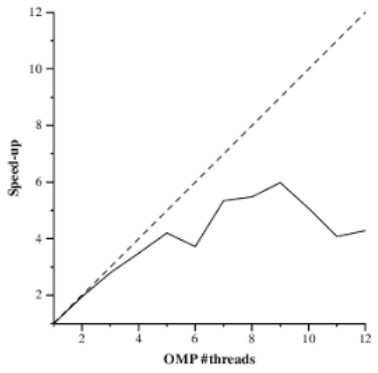
Here's the Paraver execution trace generated.



In the following plots we can see the speed-up obtained with the Jacobi implementation varying the number of threads. The speed-up improves while the number of threads executing the code is also increased. However, when the ninth thread is reached, the speed-up changes because of synchronization problems.



Average elapsed execution time



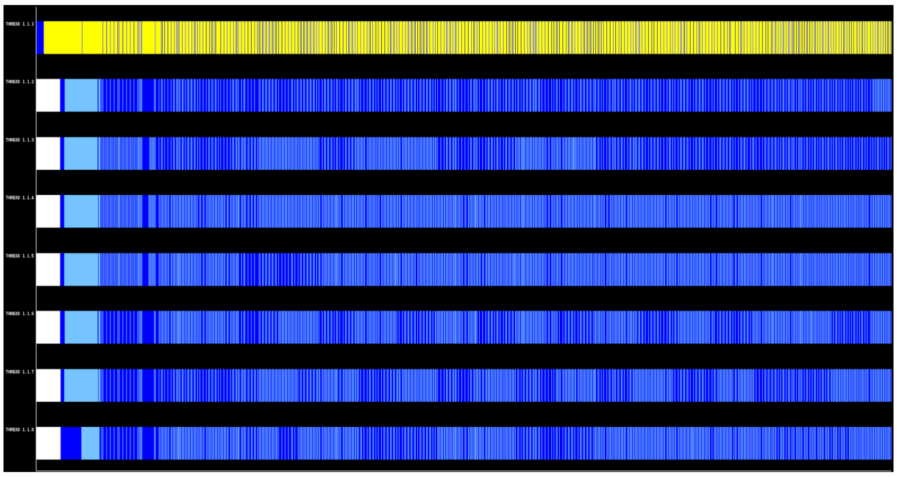
Speed-up wrt sequential time

Parallelization of Gauss-Seidel with OpenMP for

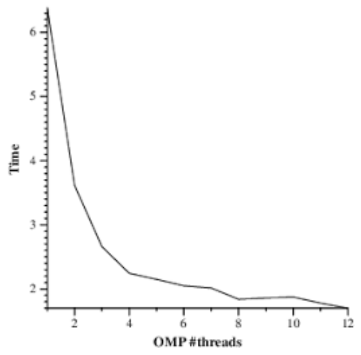
For the parallelization of Gauss-Seidel we've used the same strategy protecting the sum variable again with a reduction. In addition we've decided to make the unew variable also privative. Nevertheless, this parallelization wasn't as simple as the jacobi one, here we had to divide the matrix to give equal rows to each thread. In order to achieve the parallelization desired we needed a new vector, in this case it's processedBlocks which counts how many blocks has each thread processed.

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

    for(int i = 0; i < howmany; ++i) {
      processedBlocks[i] = 0;
    }

   int nBlocs = 32;

    #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 < nBlocs; j++){
            int jj_start = lowerb(j,nBlocs,sizey);
            int jj_end = upperb(j,nBlocs,sizey);
            if(i > 0){
                while(processedBlocks[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;
                }
            }
            ++processedBlocks[i];
            #pragma omp flush
        }
    }
    return sum;
}
```

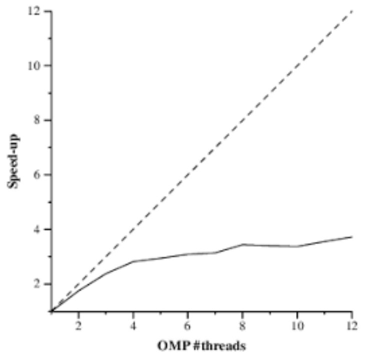
Here’s the paraver trace generated by the Gauss-Seidel program.



In the following plots we can see the speed-up obtained with the Jacobi implementation varying the number of threads. We can see that the speed-up improves as the number of threads increase. However, as it is seen in both plots, it doesn’t improve too much after 4 threads are reached.



Average elapsed execution time



Speed-up wrt sequential time

After trying diffent amount of blocks (4, 8, 16, 32, 64, ...) we have seen that the best preformance is with 32 blocks.

With more than 64 blocks there are a lot of overhead in the fork & join. With less than 8 blocks there it not an enough optimization.

The best time we obtained was with **32 blocks** (1,267 s), with more than 64 blocks and less than 8 we have an execution time over 2 s.

So we will say that the best number of blocks are 8 → 64.

4. Conclusions

The processors coherence systems may strongly affect the execution of a program, as it has been seen on the above examples. Due to this we conclude that it is an important part of the code that has to be taken into account. Additionally, we can conclude that in some cases a data decomposition strategy improves a task decomposition strategy, for example in the Gauss-Seidel program.