PAR Laboratory Assignment

Course 2018/19 (Fall semester)

Lab 5: Geometric (data) decomposition: heat diffusion equation

Pablo Vizcaino, Guillem Ramírez

User: Par2206

December 29, 2018

Contents

1 Introduction

In this laboratory assignment we will be working with heat diffusion algorithms, specially with Jacobi and Gauss-Seidel approaches.

What these algorithm does is given a number of heat focus in a 2D coordinate space, it simulates it's diffusion, getting the following results.

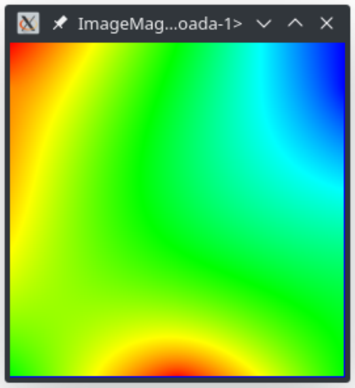


Figure 1: Heat diffusion results

First of all we will be working with Jacobi algorithm because it's parallelization it's a bit simple and we will get better in touch with heat diffusion algorithms as it's approach has differences but is similar. Later on we will introduce Gauss-Seidel approach which has a different approach.

2 Analysis of dependencies using Tareador

In this section we are going to take a look at data dependencies of both algorithm using Tareador.

2.1 Tareador analysis of Jacobi approach

First of all we worked with Jacobi algorithm. Given a first execution using tareador we got the following dependency graph.

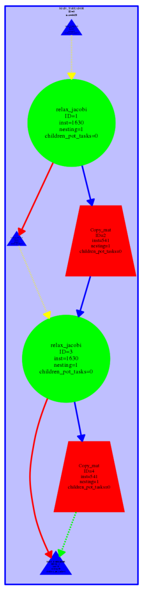


Figure 2: Jacobi Tareador default execution.

As you can see in the following image the code generated is sequential and there is much nothing we can do, as we have dependencies between the calls and copy mat. If we go for a more finner granularity to one task for the most inner loop of the algorithm we can observe the following.

You can assume that red spots are inner loop iterations. These are secuentially ordered because of data dependencies. Specially the dependencies are in the sum variable, this can be easily solved with a reduction. The other variable that causes serialization is the diff variable. You will see our approach in the next section.

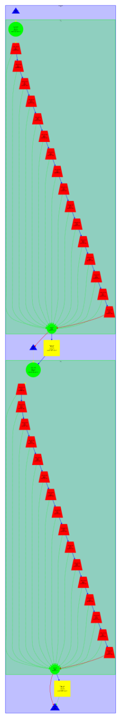


Figure 3: Jacobi Tareador finer granularity execution.

Finally, we deactivated the variables in order to see if the graph generated there is possible parallelism or not.

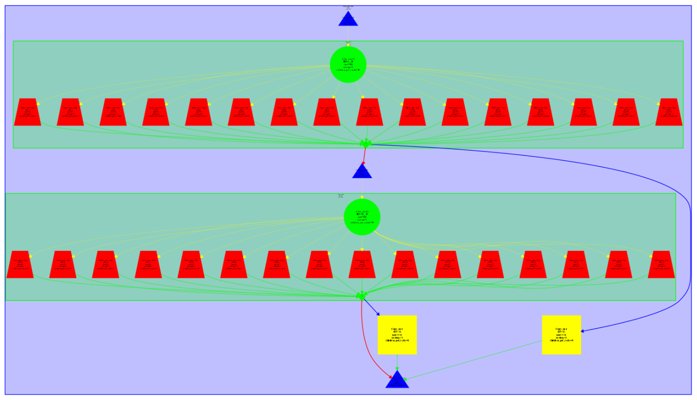


Figure 4: Jacobi Tareador finer granularity execution with sum deactivated.

As you can observe yes, the code has no dependencies between iterations.

2.2 Tareador analysis of Gauss-Seidel approach

In this subsection we are going the repeat the analysis done with Jacobi's approach.

On a first execution with no granularity set, we obtain the following task graph.

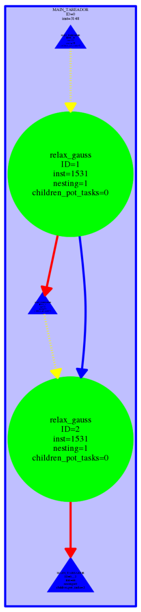


Figure 5: Gauss-Seidel Tareador default execution.

As you can see from the last subsection we get similar graph with the difference that there is not `copy_mat` between calls to the algorithm. In practice there is no actual difference as the dependencies due to the order of the code are still there, so we can not paralelize at that granularity.

Going to inner-most loop granularity we get the following graph.

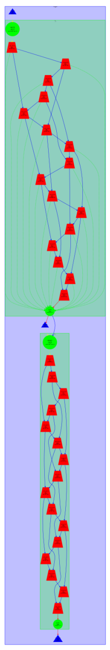


Figure 6: Gauss-Seidel Tareador finer granularity.

In comparison the the Jacobi inner most loop granularity Tareador graphic you can observe that there appear more dependencies between tasks. We have the other dependencies, for sum and diff variables, but each iteration has dependencies with $(i-1, j)$ and $(i, j-1)$ elements of the matrix, being i and j the row and the column for the current block.

Disabling the dependencies we obtain the following task graph.

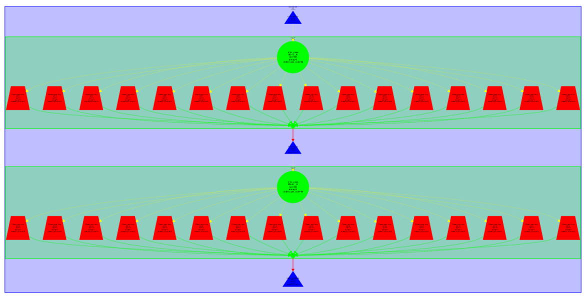


Figure 7: Gauss-Seidel Tareador finer granularity and deactivated dependencies.

So we can also paralelize the Gaus-Seidel aproach taking into account the dependencies generated.

3 OpenMP parallelization and execution analysis: Jacobi

3.1 Parallelization

The first algorithm that we parallelize is Jacobi's. In order to do that, the first thing we do is to distribute the work among the threads based on the data. We followed a geometric data block decomposition model; which means that the data is split into equal and consecutive blocks and each block is given to one processor. If the size of data can be expressed by N and the number of threads P, each block's size is N/P.

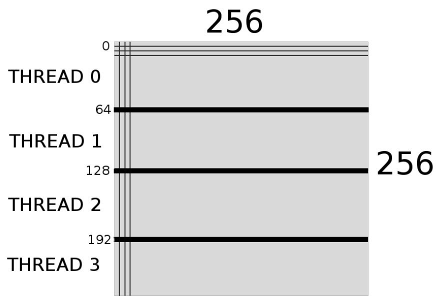


Figure 8: Diagram of geometric block data distribution

In this particular problem, we decided to create this blocks by consecutive rows. For example and as you can see in the figure, if the matrix is 256x256 elements and we have 4 threads, one will do rows 0 to 63; the next one 64 to 127, etc...

To accomplish this, we use the macros that are given to us to calculate these boundaries. When the main program arrives to the parallel region, it divdes into several threads. Since the original code defines the variable *howmany* as the number of data blocks, we keep using it instead of making as many blocks as threads. Each thread calculates its boundaries based on its thread id and then jumps into the for loops; which start and end limits are based on the boundaries calculated before. You can see this implementation on the code below.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex,
unsigned sizey)
{
  double  sum=0.0;
  #pragma omp parallel reduction(+:sum)
  {
    int howmany=4;
    if(howmany>omp_get_num_threads())howmany=omp_get_num_threads();
    //for (int blockid = 0; blockid < howmany; ++blockid) {
    int blockid = omp_get_thread_num();
    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
        double diff;
        diff = utmp[i*sizey+j] - u[i*sizey + j];
        sum += diff * diff;
          }
        }
        //}
  }
  return sum;
}
```

The variable sum encounters a data-race problem as several threads are reading and writting its value; so we decided to fix it in the easiest way possible making it a reduction. Another problem was the variable diff, which was declared in the same line as the variable sum and therefore would be shared among threads. We could have made it private in the parallel region declaration but we decided to move the variable declaration inside the region; making it local to each thread.

3.2 Performance Evaluation

Once we tested that the parallel version gives a correct output (comparing the image to the sequential one), we evaluated the speed-up. First, we compared the execution time of both sequential and parallel version and obtained that the first one lasted 5.219s with 2142.62 MFlop/s and the second one 3.235s with 3456.07 MFlop/s. This means a speed-up of 1.613.

Another measure to evaluate the performance is the strong-scallability plot.

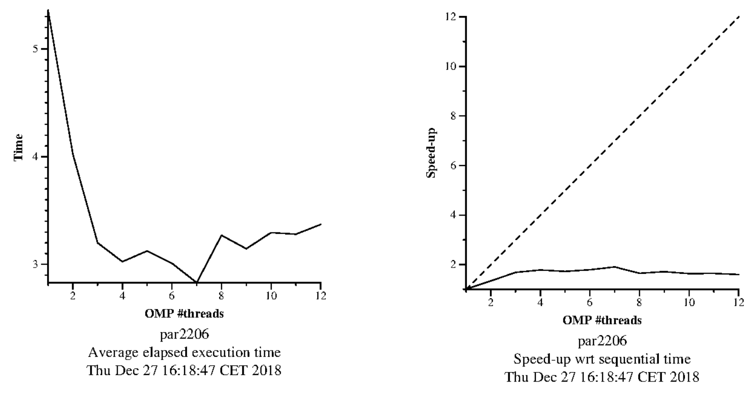


Figure 9: Jacobi strong scallabilty speed-up plots

As it can be easily seen in the figure, the scallability is really far from ideal. This is in part due to the implicit limitation in keeping the *howmany* variable. We are limiting the maximum number of blocks to 4, and in consequence only 4 threads are going to work at maximum, as you can see in the figure below where we only show one iteration (one call to the Jacobi function).

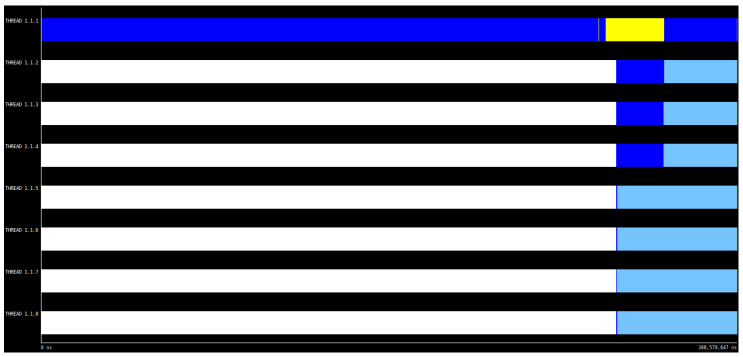


Figure 10: Work distribution on one iteration with howmany=4

It can be seen that only 4 threads are doing real work while the other 4 wait.

In order to really evaluate the potential parallelism, we are going to remove this limitation and set the number of blocks as the number of threads in the parallel region.

```
int howmany = omp_get_num_threads();
```

With this improvement, the new execution time is 2.845s and 3930.71 MFlop/s. Now the gain is 1.834 compared to the sequential version and 1.137 compared to the previous parallel version. We can observe in the following paraver capture that we fixed the problem with useless threads.

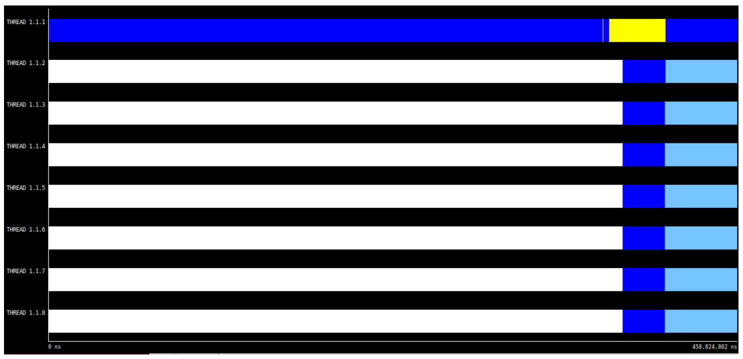


Figure 11: Work distribution on one iteration with howmany=omp_get_num_threads()

This improvement is less notable when looking at the strong scallability plots, which are still far from ideal.

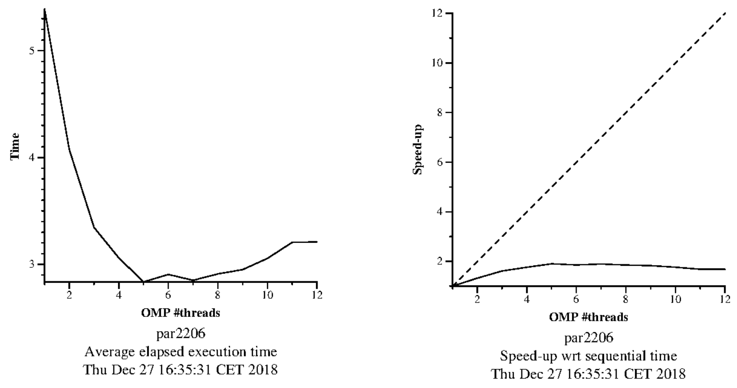


Figure 12: Jacobi improved version strong scallabilty speed-up plots

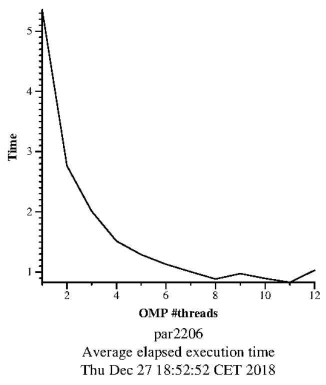
There's still more room for improvement in the parallelization of the Jacobi algorithm. We can see that each iteration in the main code also calls the funciton *copy_mat*. We decided to parallelize this function too following the same decomposition as the *Jacobi* function :

```
void copy_mat(double *u, double *v, unsigned sizex, unsigned sizey)
{
  #pragma omp parallel
  {
    int howmany = omp_get_num_threads();
    int blockid = omp_get_thread_num();
    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++)
        v[ i*sizey+j ] = u[ i*sizey+j ];
  }
}
```

After executing this new version, we obtain an execution time of 0.852s and 13129.18 MFlop/s. This represents a gain of 6.13 compared to the sequential version (5.219s) and a gain of 3.34 compared to the last parallel version (2.845). We can put all three previous results in a table to have a better look at the improvements.

Version	Execution Time (s)	Mflops	gain from sequential
Sequential	5.219	2142.62	1
Howmany=4	3.235	3456.07	1.613
Removed limitation	2.845	3930.71	1.834
Copy_mat parallel	0.852	13129.18	6.13

This great improvement can also be seen in the following strong-scallability plots:



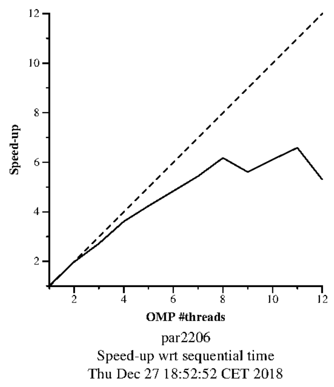


Figure 13: Jacobi final improvement version strong scallabilty speed-up plots

4 OpenMP parallelization and execution analysis: Gauss-Seidel

4.1 Parallelization

In this section we are going to parallelize the Gauus-Seidel algorithm. The parallelization is going to be different since the dependencies are different, as we seen on the first section. In this case, there are dependencies between iterations inside the Gauss function, more specifically with adjacent positions of the matrix *u*.

We also have to do the same adjustments to the variable sum and diff, with the addition of a new one called unew. The first one is going to be a reduction and the last two are going to be private for each thread.

In this algorithm we first used the #pragma omp for ordered construct, which assures the correct execution of nested loops in order. This is needed because we will also be using the depend clause in order to indicate the dependencies between the matrix. First, we use the sink clause to mark the dependencies to previous iterations and then we use the source clause to indicate that the corresponding iteration has finished its work.

Here you can see the code with all the pragmas:

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
  double unew, diff, sum=0.0;
  #pragma omp parallel for ordered(2) reduction(+:sum)
                                  private(diff,unew)
  for (int i=1; i<= sizex-2; i++) {
    for (int j=1; j<= sizey-2; j++) {
      #pragma omp ordered depend (sink:i,j-1) depend (sink:i-1,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;
}
```

4.2 Performance evaluation

Just looking at the execution time of this version is enough to tell that this version is far from ideal. The sequential code got an execution time of 6.279s and 1402.62 MFlops and the parallel one 95.355s and 92.35 Mflops. This means a decrease of 1418.63%; a really terrible parallelization. This is due to the overhead given by the dependencies.

The execution lasts so long that it isn't viable to show paraver captures or strong plots, but it's easy to imagine how terrible they would be considering the increase of the execution time. We could change the granularity in order to maybe get a better performance, but we decided to try to implement another type of parallelization instead since the first method has already been used in the past lab assignments.

4.3 New approach

In order to parallelize this algorithm we are going to need a really different approach. Instead of using the #pragma omp for clause we are going to implement a data decomposition similar to the one in Jacobi's algorithm. This means applying a geometric block data decomposition. First, we are going to implement it the same way as in Jacobi's algorithm and see what problem we encounter.

First, we are going to use the following code:

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
  double unew, diff, sum=0.0;
  int howmanyx = omp_get_num_threads();
  #pragma omp parallel for ordered(1) reduction(+:sum)
                                  private(unew) private(diff)
  for (int blockidi = 0; blockidi < howmanyx; ++blockidi) {
    int i_start = lowerb(blockidi, howmanyx, sizex);
    int i_end   = upperb(blockidi, howmanyx, sizex);
    #pragma omp ordered depend(sink:blockidi-1)
    for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
      for (int j=1; j<= sizey-2; 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(sink:source)
  }
  return sum;
}
```

With this version, we obtain an execution time of 6.763 and 1302.18 Mflops. This is still a decrease in execution time (7.71%) but much better than the previous version. The execution is actually like the serial one, since each block has to wait until the previous one has finished to start. The extra time is due to OpenMP overheads.

Making a simple block decomposition by rows makes us have to wait until all the computation of the previous block has finished until the next one can start. This is caused by the dependencies that are in the borders of adjacent blocks. To improve this, we are also going to create blocks on the ”j” dimension.

In order to define the size of this blocks, the ideal case would be to have a number of threads with a natural root, so we can have *sqrt(omp_get_num_threads())* blocks in each dimension. Since we can’t assure that this will be the case, we thought of a different approach. The number of blocks in the ”i” dimension will be the integer floor of the square root of the number of threads; and the number of blocks in the ”j” will be the floor of the number of threads divided per the square root. Doing this we get the maximum number of blocks evenly distributed among all threads, with each thread doing maximum one block.

In the following figure you can see an approach of how the data distribution can end. Dependencies are between a block and the one to its left and the one over it.

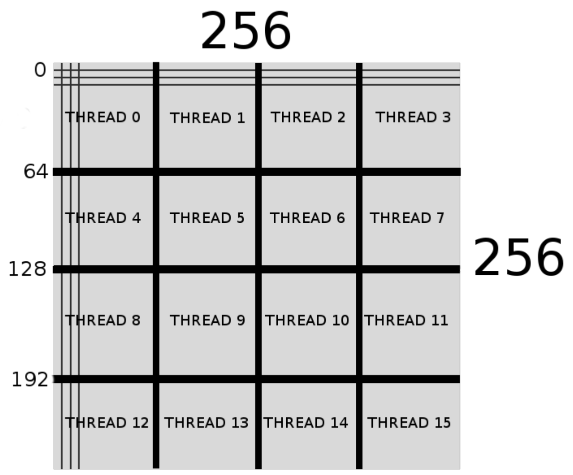


Figure 14: Better block distribution

The code is the following:

```
double relax_gauss (double *u, unsigned sizex , unsigned sizey)
{
  double sum=0.0;
  #pragma omp parallel
  {
  int P = omp_get_num_threads();
  int root = sqrt(P);
  int howmanyi=root;
  int howmanyj=P/root;
  int id = omp_get_thread_num();
  int id_i = id/howmanyj;
  int id_j = id%howmanyj;
  int i_start = lowerb(id_i, howmanyi, sizex);
  int i_end   = upperb(id_i, howmanyi, sizex);
  int j_start = lowerb(id_j, howmanyj, sizey);
  int j_end   = upperb(id_j, howmanyj, sizey);
  #pragma omp task depend(out:u[i_end*sizey + j_end])
              depend(in:u[sizey*(i_start-1)+j_end])
               depend(in:u[sizey*i_end+j_start-1])
  {
  double tmp = 0.0;
  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++) {
      double 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
      double diff = unew - u[i*sizey+ j];
      tmp += diff*diff;
      u[i*sizey+j]=unew;
    }
  }
  #pragma omp atomic
  sum += tmp;
  }
  }
  return sum;
}
```

When trying to evaluate this code, the result wasn't correct and we weren't able to fix it. We also tried the following code, but it didnt work. The result was really close to the sequential one, but not quite. We haven't been able to find what is causing it or another way to write the algorithm.

```
struct abc{
        double sum;
        int done;
};
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
  double sum=0.0;
  struct abc thread_vector[omp_get_max_threads()];
  for(int i=0;i<omp_get_max_threads();++i) {
    thread_vector[i].sum = 0.0;
    thread_vector[i].done = 0;
  }
  #pragma omp parallel
  {
  int P = omp_get_num_threads();
  int root = sqrt(P);
  int howmanyi=root;
  int howmanyj=P/root;
  int id = omp_get_thread_num();
  int id_i = id/howmanyj;
  int id_j = id%howmanyj;
  int i_start = lowerb(id_i, howmanyi, sizex);
  int i_end   = upperb(id_i, howmanyi, sizex);
  int j_start = lowerb(id_j, howmanyj, sizey);
  int j_end   = upperb(id_j, howmanyj, sizey);
  while((id>howmanyj && thread_vector[id-howmanyj].done==0) ||
                  (id_j>0 && thread_vector[id-1].done==0)){
    __asm__("nop");
  }
  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++) {
      double 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
      double diff = unew - u[i*sizey+ j];
      thread_vector[id].sum += diff*diff;
      u[i*sizey+j]=unew;
    }
  }
  thread_vector[id].done=1;
  }
  for(int i=0; i<omp_get_max_threads(); ++i)
    sum+=thread_vector[i].sum;
  return sum;
}
```

5 Conclusions

In this laboratory assignment we took a loop at data decomposition approach which gave us a different point of view when parallelizing a program.

On the Jacobi algorithm we saw how important is it to not limit the parallelization arbitrarily and also to parallelize all the posible code (p.e the copy_mat function)

We also learned that there are trade-offs between parallelization, i.e when a lot of tasks are created we express dependencies as they are, but we loose a lot of performance due to overheads.

Other things that we learned is that there is not a perfect solution for everything, in the first session we thought that we could find out a perfect solution with a good scallability, after a few hours of work we learned that not everything is perfectly parallelizable.

Another problem that we encountered is trying to do a data parallelization where there's dependencies between data blocks. There are many ways to go over it, but most of them cause a big overhead.