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

Genís Bosch
Oriol Fonollà
Curs 2017/2018 Q2
Par2202

Introduction

In this laboratory assignment, we are going to practice and observe different parallelization strategies on a sequential code that simulates heat diffusion in a solid body. We'll see two different solvers for the heat problem, Jacobi and Gauss-Seidel.

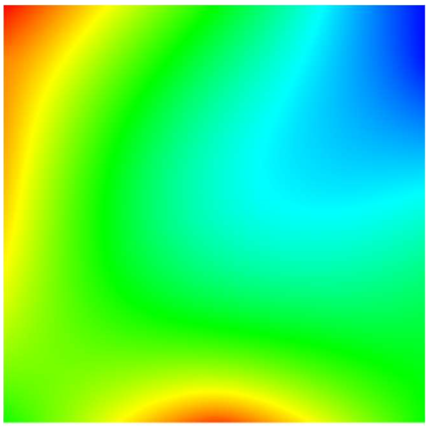


Figure 1. Heat representation

First, we'll use Tareador to see the potential parallelization. On both algorithms, we'll place the tareador_start_task on the innermost loop of the function, like we can see in the code:

```
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");
            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");
        }
     }
   }

   return sum;
```

}

As we place the tareador_tasks, we realise that the variable sum is common in all the cells, so we'll have to place a tareador_disable_object to disable the sum variable on the Tareador execution. On the OpenMP parallel code, to solve this issue, we'll probably use a reduction clause.

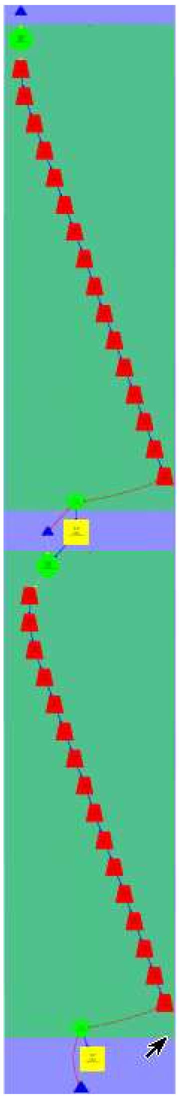


Figure 2. Jacobi-Tareador

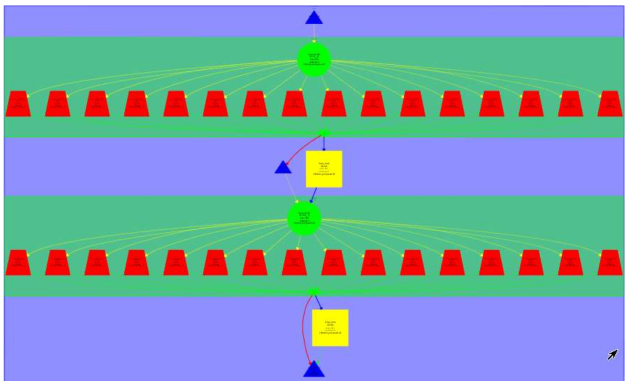


Figure 3. Jacobi-Tareador with sum disabled

As we can see in the Figure 2, it’s obvious that there’s a dependence over the variable sum, so to explore the real potential parallelization we’ll have to look at Figure 3, where the variable sum has been disabled by the Tareador clause tareador_disable_object.
With the Gauss-Seidel code we are going to apply the exact same strategy, as we’ll use the Tareador calls to see the potential parallelism, and we’ll place those in the innermost loop of the function. We’ll also have to disable the sum variable, for the same reason we explained before.

```
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");
          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");
      }
    }
  }

  return sum;
}
```

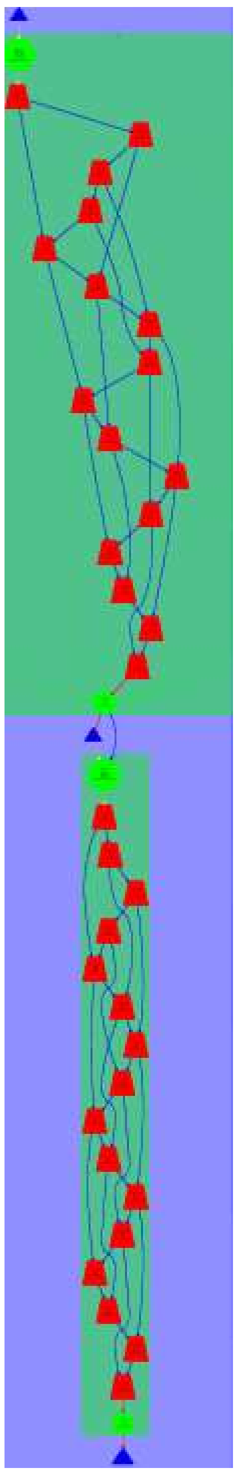


Figure 4. Gauss-Tareador

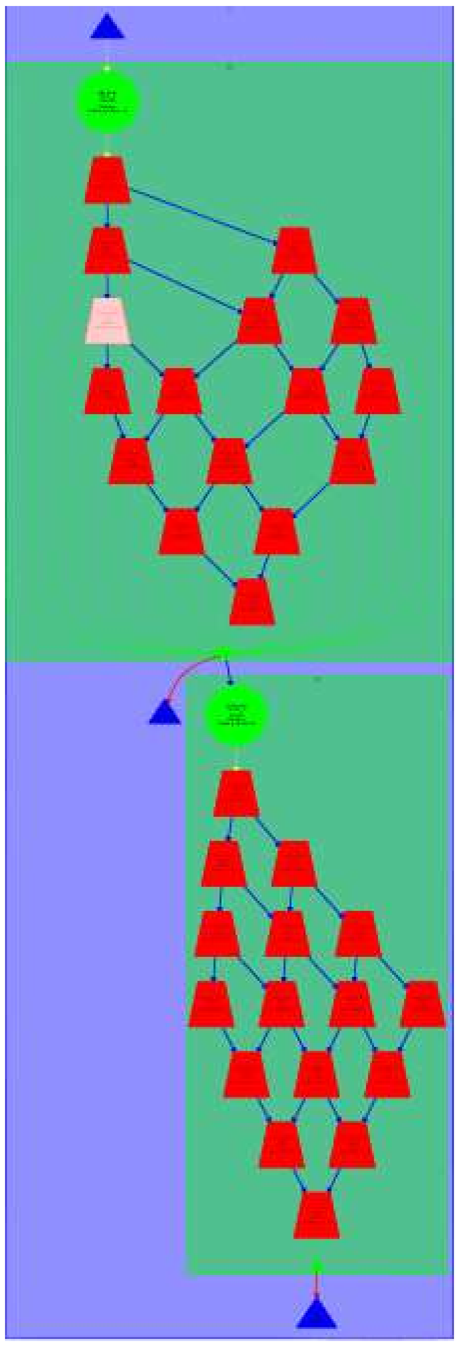


Figure 5. Gauss-Tareador with sum disabled

As we can see in the Figure 4, if we don't disable the sum variable, there's parallelization but with a lot of dependencies. If we take a look at Figure 5, where we have disabled sum, the dependencies are much clear and we can see that one task just depends on the upper and left tasks.

Parallelization strategies

We'll be trying two algorithms and its parallelization, Jacobi and Gauss-Seidel:

The first one saves the result of the computation of each cell into an auxiliary matrix, and then uses this new one to compute the other variables, like diff and sum.
Instead, the Gauss-Seidel strategy computes the element of each cell and saves the result into its own cell. This makes the strategy more efficient but increases the synchronization, as one cell depends on the upper and the left ones to compute itself.

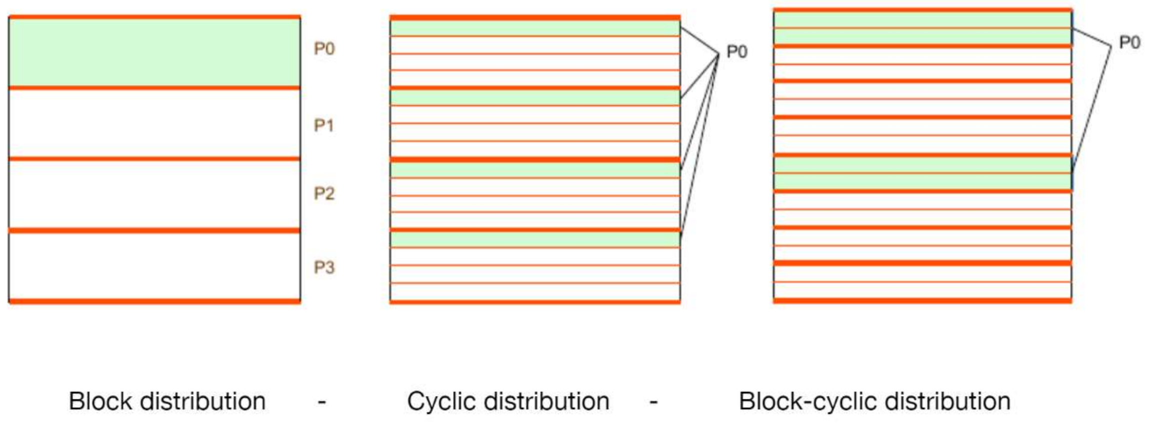

P0
P1
P2
P3
P0
P0
Block distribution - Cyclic distribution - Block-cyclic distribution

Figure 6. Data decomposition strategies

For the parallelization strategies, we'll apply data decomposition into the algorithms. There are 3 types of data decomposition as we can see in the Figure 6: by blocks, cyclic and block-cyclic. In this session, on both algorithms we'll be using blocks distribution, but we'll change the block size depending on our needs.

Jacobi

After analysing the tasks in Tareador, we decide to implement the parallelism with one pragma omp for clause on the blockid loop to give a thread for each block.
The code will look like this, as we'll place a pragma omp for clause with a private clause for the diff variable and a reduction clause for the sum variable.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{

  double diff, sum=0.0;
   int howmany=8;

#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;
}
```

Gauss-Seidel

In this algorithm, our parallelization will be based on blocks, but instead of just giving blocks to divide the i loops, we'll also divide the j loops. We'll get this by nesting the two blockid loops and placing an ordered clausule on them, so the blocks are computed the way we want.

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

   int howmany=4;

         #pragma omp parallel for ordered(2) private(diff)
   for (int blockid = 0; blockid < howmany; ++blockid) {
         for (int blockidj = 0; blockidj < howmany; ++blockidj) {
                  int i_start = lowerb(blockid, howmany, sizex);
                  int i_end = upperb(blockid, howmany, sizex);

                  int j_start = lowerb(blockidj,howmany,sizey);
                  int j_end = upperb(blockidj,howmany,sizey);


                  #pragma      omp      ordered      depend(sink:      blockid-1,blockidj)
depend(sink:blockid,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;
}
```

Performance evaluation

To prove and evaluate different parallelization strategies we used three different scripts: submit-omp.sh, submit-strong-omp.sh, submit-i.sh

Submit-strong-omp: is used for analyze the scalability of the program by creating two speed–up plots (complete application and multisort only).
Submit-omp : used to submit the Multisort program on the Boada processors, to know if the code is right and the execution time of each program.
Submit-i: used to submit the program on the Boada processors and create a paraver trace to analyze.

Jacobi
Using that tools in the Jacobi parallelization strategy we've obtained the following plots and paraver trace.

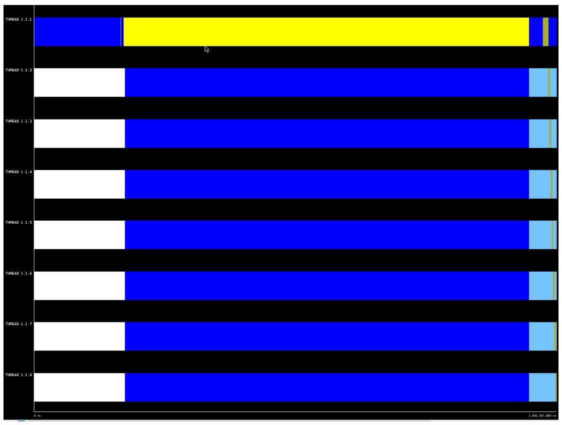
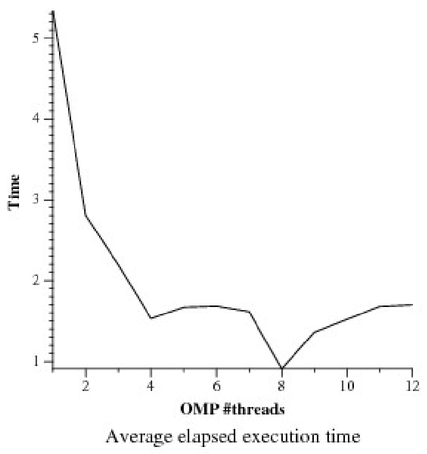


Figure 7. Jacobi Paraver trace



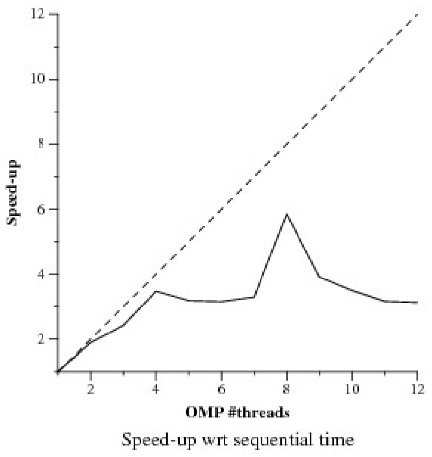


Figure 8. Jacobi plots

In the Figure 8 plots we can see the speed-up obtained with the Jacobi implementation when we change the number of threads. It improves while the number of threads is increased. However, when the eight thread is reached, the speed-up changes because of synchronization problems.

Gauss-Seidel

Using that tools in the Jacobi parallelization strategy we've obtained the following plots and paraver trace.

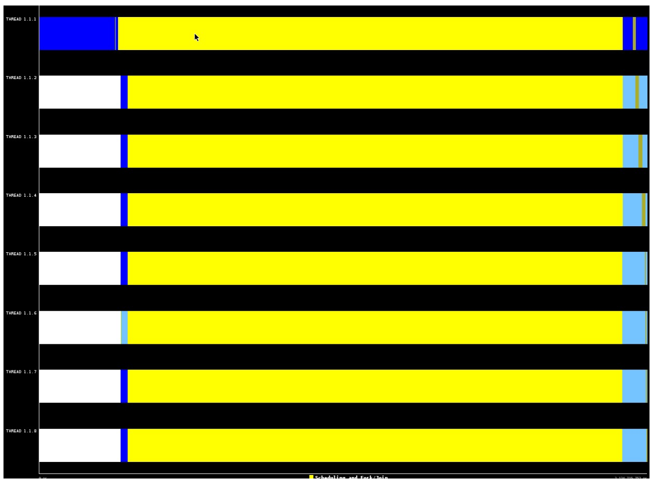
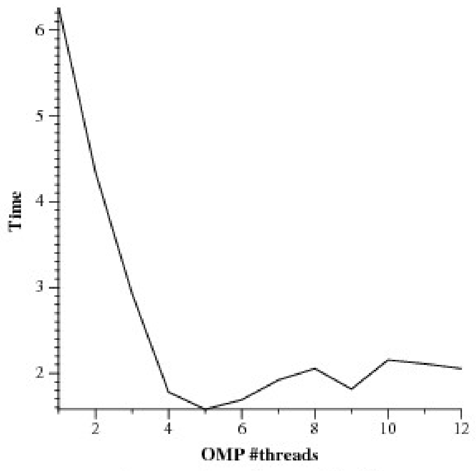
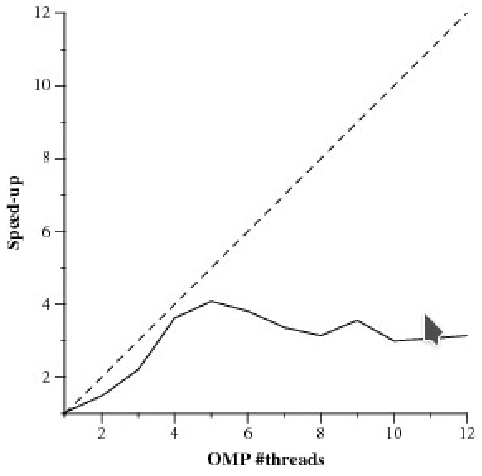


Figure 9. Gauss-Seidel Paraver trace



Average elapsed execution time



Speed-up wrt sequential time

Figure 10. Gauss-Seidel Paraver trace

In the Figure 10 plots we can see the speed-up obtained with the Gauss-Seidel implementation when we change the number of threads. It improves while the number of threads increases. However, as it is seen in both plots, it doesn't improve too much after 4 threads are reached.

Conclusions

After performing this laboratory we can say that the processors coherence systems affect the execution of a program. That's why we think that it is an important part of the code that has to be analysed and prepared. In some cases, this coherence systems can be accompanied with data decomposition strategy in order to improve it performance.