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

par4111
Adrià Cabeza, Xavier Lacasa
Departament d' Arquitectura de Computadors

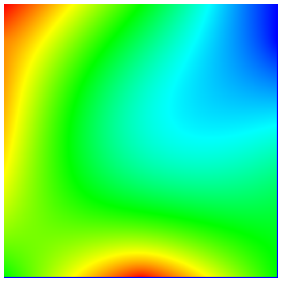
June 8, 2019
2018 - 19 SPRING

Contents

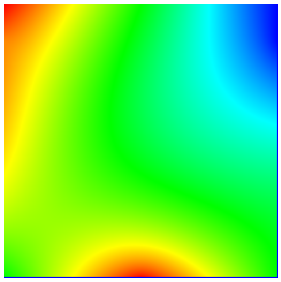
1 Introduction

In this laboratory session we will be dealing with a real problem and we will try to parallelize a piece of code using the most convenient task strategy.

All our different parallelization strategies will be made on a sequential code called *heat.c*. This code simulates heat diffusion in a solid body using two different solvers for the heat equation: **Jacobi** and **Gauss-Seidel**.



(a) Heat simulation using Jacobi



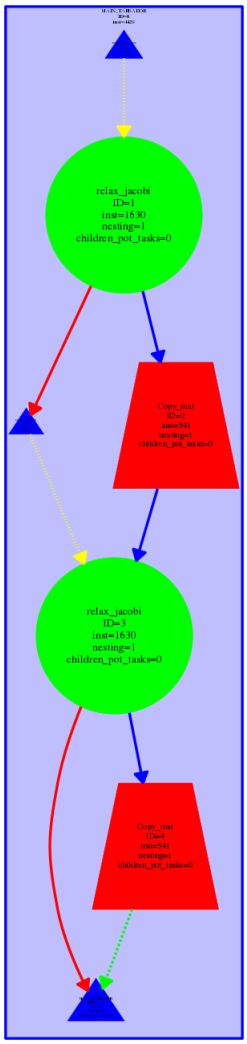
(b) Heat simulation using Gauss

2 Analysis of task granularities and dependences

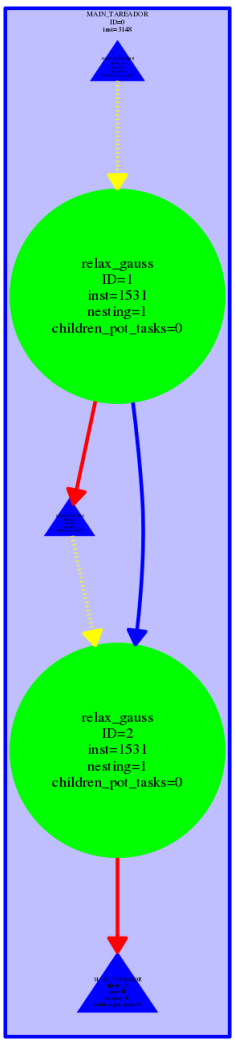
In this section we will use *Tareador* to analyse the task graphs generated when using the two different solvers. These dependency graphs will be really useful in order to implement the proper parallelization strategy.

Firstly we have used Tareador to see the dependency graph of the versions that were already given, for the **Jacobi** and the **Gauss-Seidel** solvers.

The parallelization strategy used for both solvers was to generate the finest grain tasks possible. To archieve it we have created a task inside the innermost loop of both methods.
After implementing the previous strategy we checked the Tareador to see the dependency graph of each solver.



(a) Jacobi dependencies



(b) Gauss-Seidel dependencies

2.1 Jacobi

Code for the innermost loop parallelization strategy:

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

    int howmany=4;
    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];
        sum += diff * diff;
        tareador_end_task("inner_loop_jacobi");
    }
      }
    }

    return sum;
}
```

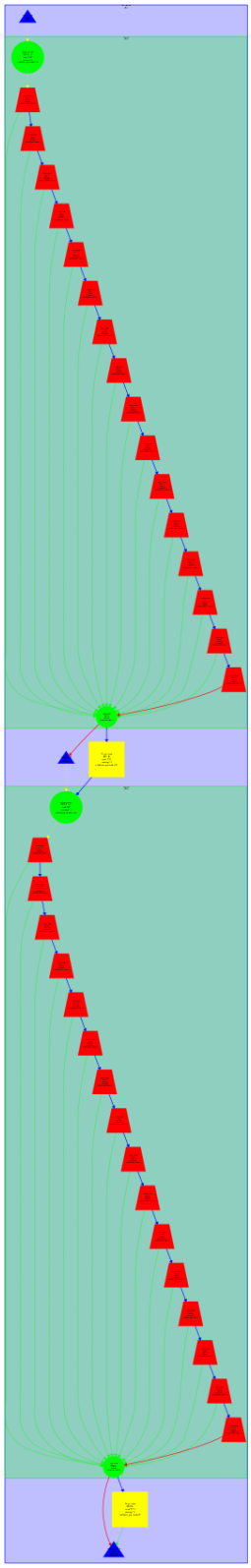


Figure 3: Dependency graph for Jacobi

We have observed that there is dependency over the sum variable: it is causing serialization of the tasks. If we ignore it from the analysis, please see figure 5, the dependency changes completely (see in Figure 4 and we obtain a completely flat dependency graph, which means that the only dependence in the program was that variable.

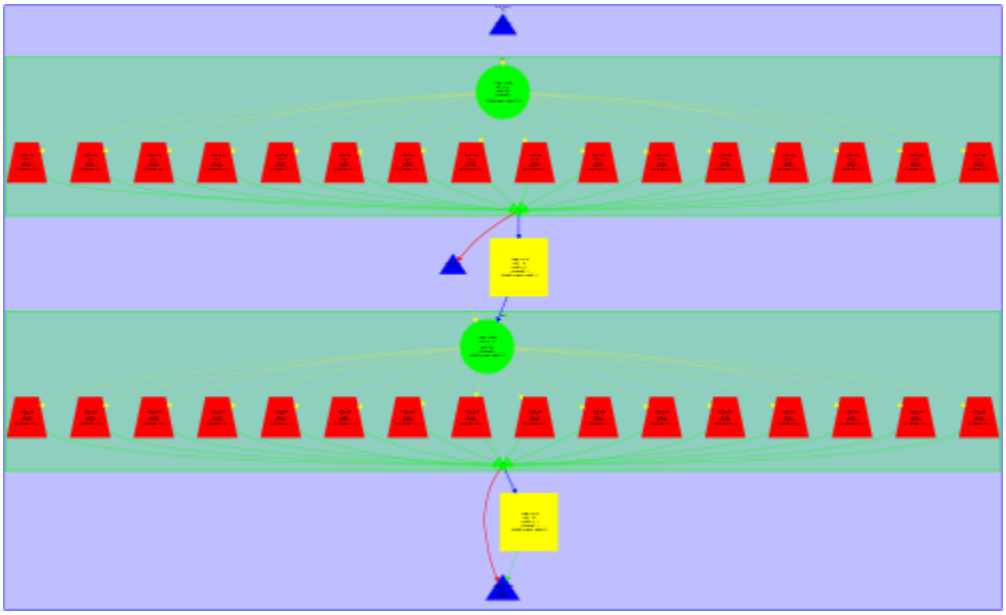


Figure 4: Dependency graph for Jacobi without sum variable

```
...
tareador_disable_object(&sum);
sum += diff * diff;
tareador_enable_object(&sum);
...
```

Figure 5: Code snippet to disable the variable sum

2.2 Gauss-Seidel

Code for the innermost loop parallelization strategy:

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

    int howmany=4;
    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];
      sum += diff * diff;
      u[i*sizey+j]=unew;
      tareador_end_task("inner_loop_gauss");
       }
     }
   }
   return sum;
}
```

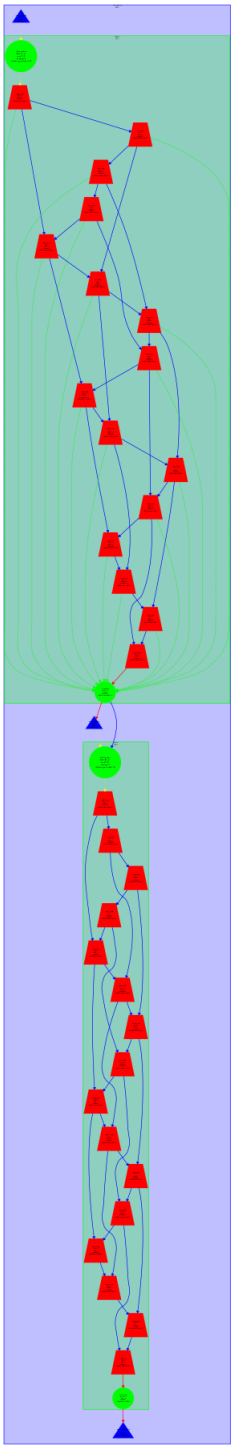


Figure 6: Dependency graph for Gauss-Seidel

After implementing the innermost loop task stategy we have used *Tareador* to see the dependency graph and we have observed some dependencies. Like in the previous version (Jacobi), we have disabled the sum variable (adding the piece of code in figure 5) to see if that variable was the only

one provoking dependencies. However we can see that we still have dependencies (see 7).

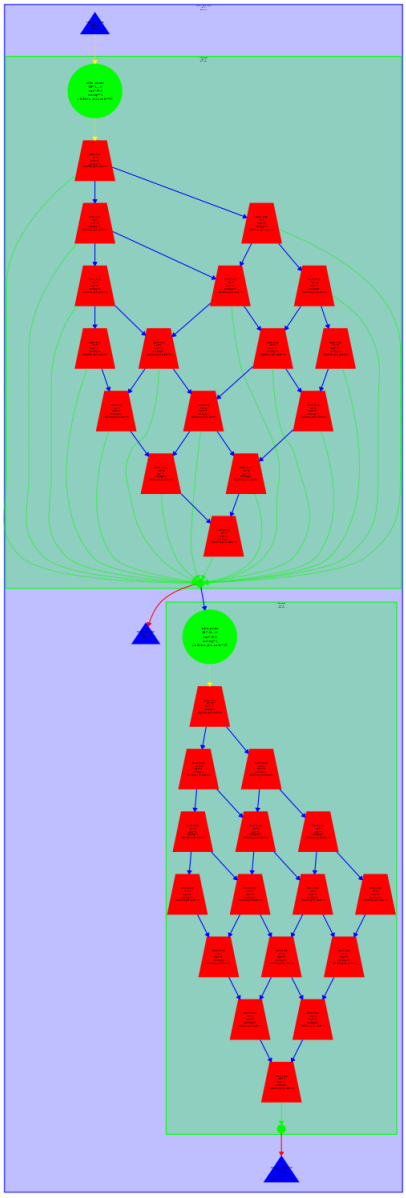


Figure 7: Dependency graph for Gauss-Seidel without variable sum

If we had to guarantee these dependences using *OpenMP* parallelization based on **#pragma omp for** we would do it using a **reduction construct**. Using this construction we would be able to prevent sequential execution. This solution stores the partial results in a vector and after the parallel region makes the sum of the partial results of each threads. However several other techniques could have been applied like using a critical pragma combined with a shared variable sum, using a pragma atomic instead of critical, etc.

3 OpenMP parallelization and execution analysis: *Jacobi*

Based on the dependencies analysis we have already made, we will paralellize it using **#pragma omp** constructions. However, we will not implement the maximum granularity because that would cause too much overhead. In order to solve it we must apply a data decomposition. First we have to choose between the input or output data structures that must be decomposed. Then, this structure will be partitioned across tasks following a data distribution: block, cyclic or block-cyclic.

In this case the solution must group consecutive pixels, that is to say, use blocks; being $block\_size = n\_rows \div n\_threads$, so each thread does a block. With **Jacobi** we have a table of size N and we divide N by the number of threads obtaining this:

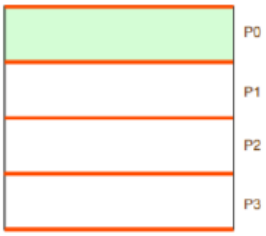


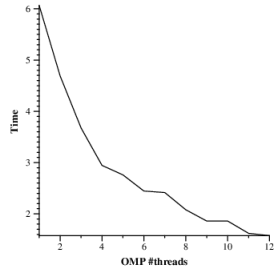
Figure 8: Jacobi data decomposition

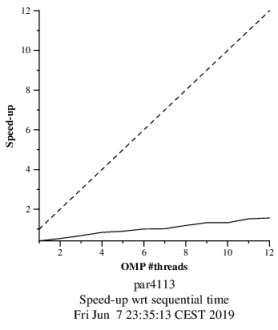
Down below, you will find the code for the Jacobi implementation. We used **#pragma omp parallel for** in the first loop. Moreover, as we said before, in order to solve the sum dependency, we will use a **reduction**. We have also noticed that the variable **diff** has dependencies, so we have solved it too.
After implementing everything, we noticed that we had a serialization because of the matrix so we had to paralellize it too.

```
/*
 * Function to copy one matrix into another
 */
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for collapse(2) 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 ];
}

/*
 * Blocked Jacobi solver: one iteration step
 */
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    #pragma omp parallel private (diff) reduction(+: sum)
    {
    int blockid = omp_get_thread_num();
    int howmany=omp_get_num_threads();
```

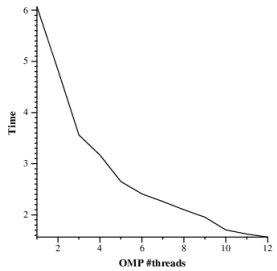
```
    int i_start = lowerb(blockid, howmany, sizex);
    int i_end = upperb(blockid, howmany, sizex);
    for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
      for (int j=1; j<= sizey-2; j++) {

      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;
}
```

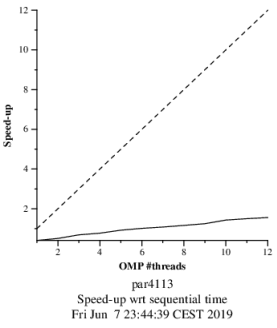
Here we can see the scalability of two versions: the one that has not parallelized the *copymat* function and the one we have given, which parallelizes everything:





(a) Scalability plot for Jacobi without parallelizing the matrix

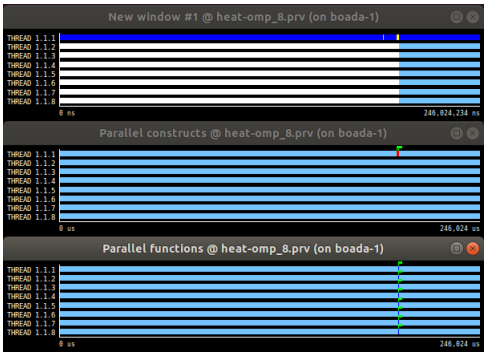




(b) Scalability plot for Jacobi

We can see that the speed-up obtained with the Jacobi implementation improves while the number of threads executing the code is also increased. However when the ninth thtread is reached,

the speed-up changes because of synchronization problems. Further parallelization analysis is shown in the respective Paraver traces:



(a) Paraver traces for Jacobi without parallelizing the matrix

(b) Paraver traces for Jacobi

4 OpenMP parallelization and execution analysis: *Gauss-Seidel*

In this section we are going to parallelize *relax_gauss*, in order to tranform the sequential execution into a parallel one. To do so, we have to do locking by column. Add a new loop and implement the synchronization of the blocks.

```
/*
 * Blocked Gauss-Seidel solver: one iteration step
 */
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(unew, diff) reduction(+:sum)
    for (int blockid_row = 0; blockid_row < howmany; ++blockid_row) {
      for (int blockid_col = 0; blockid_col < howmany; ++blockid_col) {
        int i_start = lowerb(blockid_row, howmany, sizex);
        int i_end = upperb(blockid_row, howmany, sizex);
        int j_start = lowerb(blockid_col, howmany, sizey);
        int j_end = upperb(blockid_col, howmany, sizey);

        #pragma omp ordered depend (sink: blockid_row-1, blockid_col)
        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;
}
```

In the following plots, we can see the speed-up obtained with the Gauss-Seidel 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 does not improve too much after 4 threads are reached.

Here we can see the scalability plot

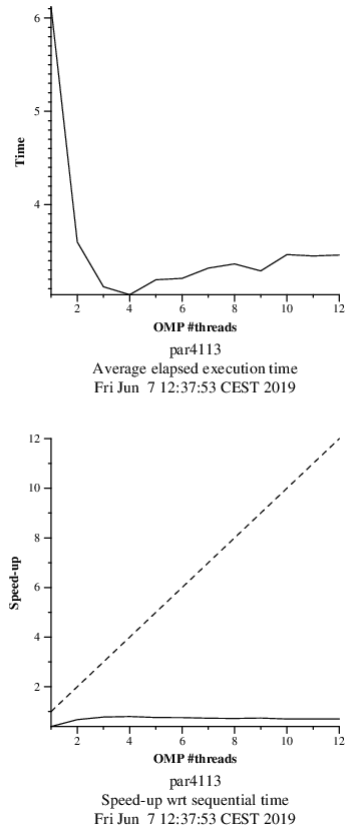


Figure 11: Scalability plot for Gauss-Seidel

As you may have noticed, parallelize the **Gauss-Seidel** function is more complicated than the Jacobi one. That is because we find dependencies between iterations and to solve it we have implemented a block distribution as opposed to the row decomposition we had initially implemented. By

using this kind of decomposition we can avoid the dependency that was generated in the *i-1* and *j-1* elements.

Paraver traces look as follow:

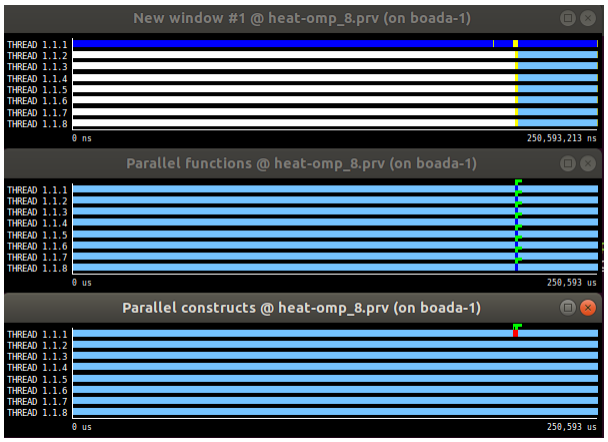


Figure 12: Paraver traces for Gauss-Seidel

5 Conclusion

In this laboratory we have been able to parallelize a complex code of two equations solvers for a real world problem. We have also experimented what can happen in performance whenever there are dependencies and we learned to solve it.

We can conclude that in the this subject we have learned from basic to more advanced concepts about parallelization using good and prepared material, like boada or paraver. It has been a good experience and we have learnt a lot in those laboratory sessions.