LAB 5: DELIVERABLE

Geometric (data) decomposition:
solving the heat equation

Jacobo Moral
Carles Pàmies
PAR4101
Q1 2017/2018

INDEX

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?

```
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("InnerForRelaxJacobi");
            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("InnerForRelaxJacobi");
        }
      }
    }

    return sum;
}
```

Figure 1: relax_jacobi function with tareador API

We can see the calls to tareador API in *Figure1*. Specifically we can see that we create a task in the innermost loop of the function *relax_jacobi* at the same time we disable variable *sum* to deny access to it from all threads at the same time.

We can see the effect of disabling this variable in figures *Figure2* and *Figure3*, without disabling *sum* and disabling the variable, respectively.

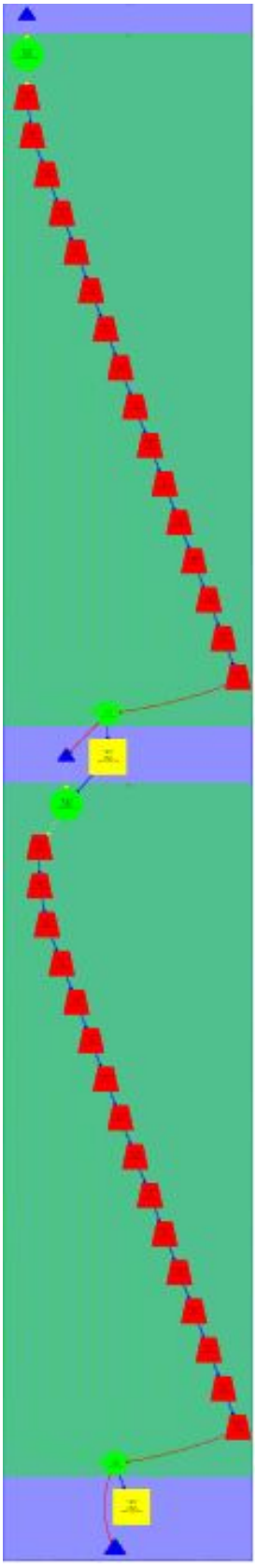


Figure 2: dependency graph of jacobi without disabling sum variable

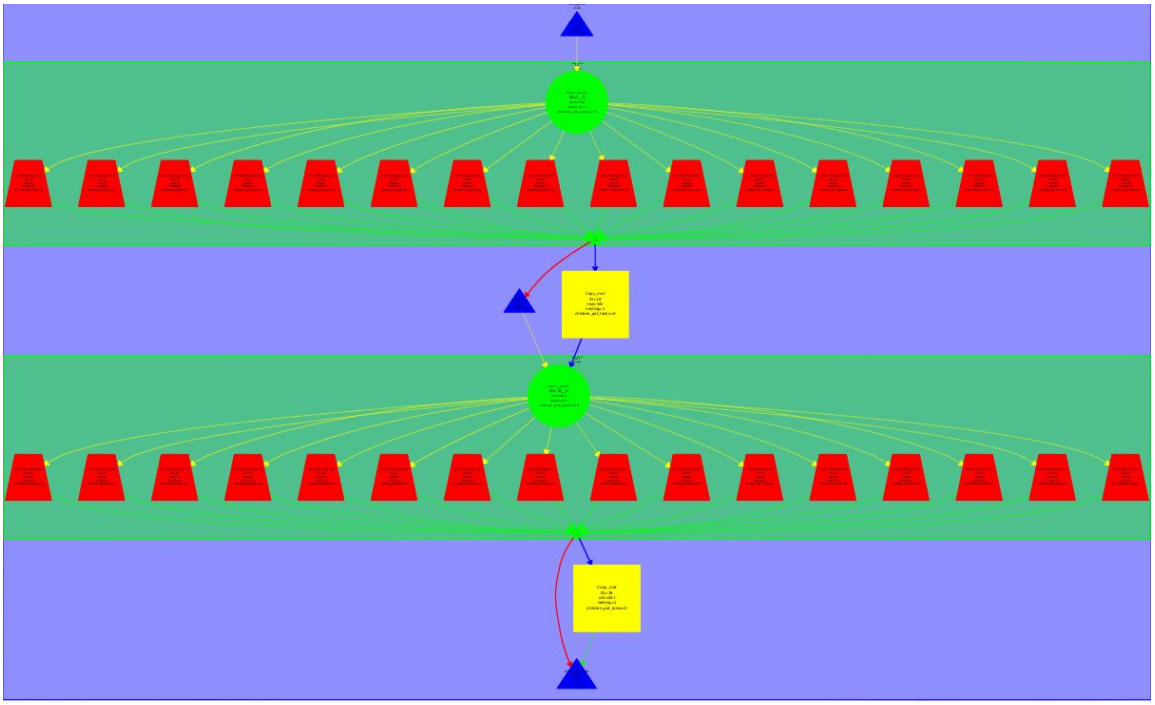


Figure 3: dependency graph of jacobi disabling sum variable

In the other hand, for Gauss function we have a similar problem. We have created a task in the innermost loop (*Figure 4*). And, just like before, we had to disable variable *sum* as we can see in the code below.

```
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("InnerForRelaxGauss");
            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("InnerForRelaxGauss");
        }
      }
    }

    return sum;
}
```

Figure 4: relax_gauss function with tareador API

We can see the effect of disabling variable *sum* in figures *Figure4* and *Figure5*, without disabling *sum* and disabling the variable, respectively.

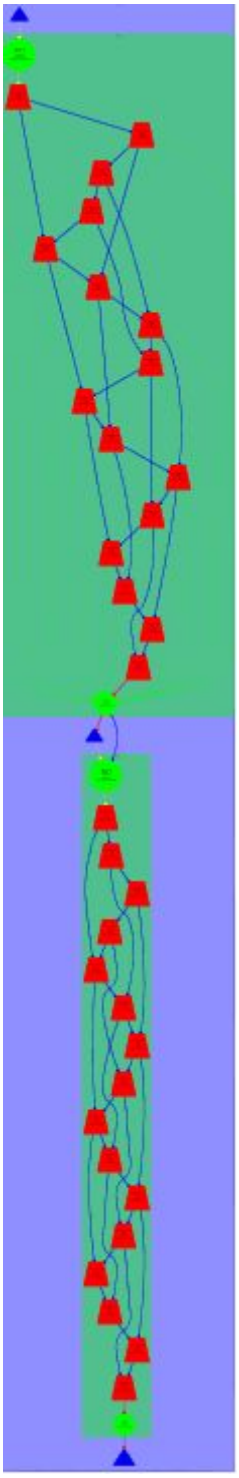


Figure 4: dependency graph of gauss without disabling sum variable

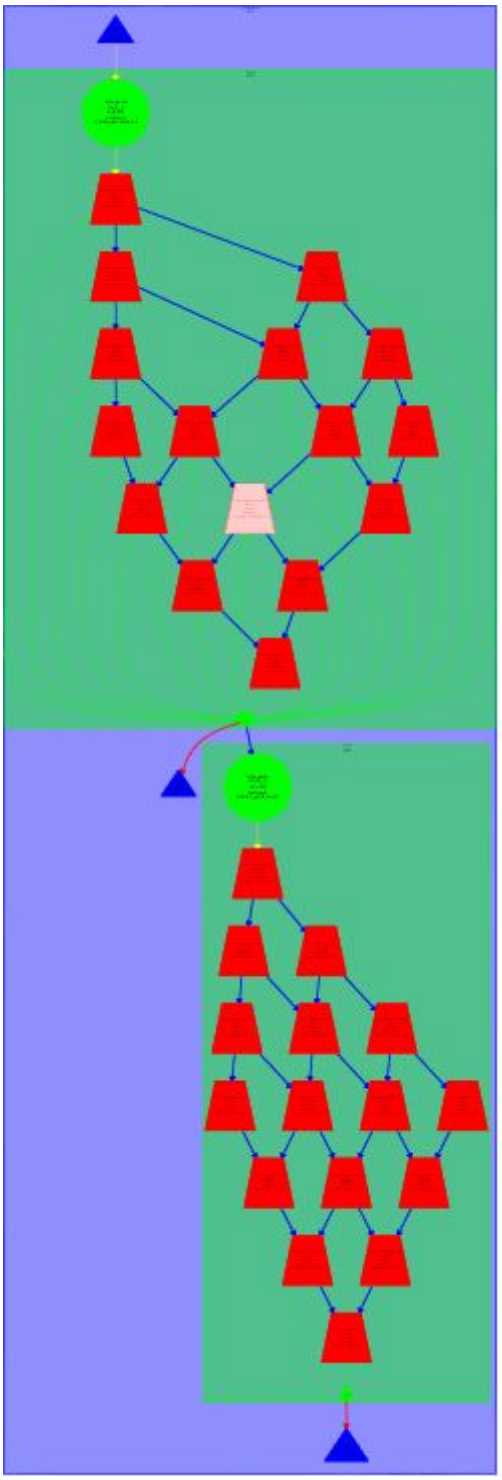


Figure 5: dependency graph of gauss disabling sum variable

In both cases we can protect the variable sum with clause *reduction*, as shown below:

*#pragma omp for **reduction** (+:sum)*

For clause in order to distribute iterations among all threads.

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.

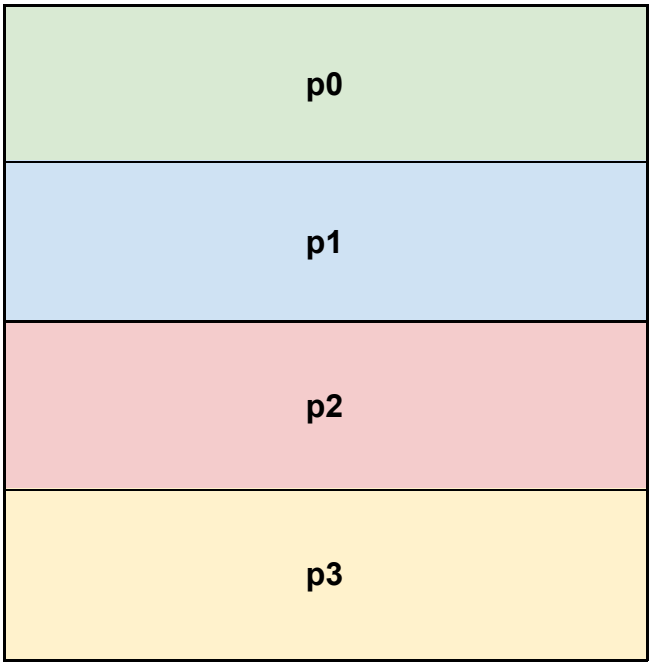


Figure 7: Data decomposition

The data each processor executes is equal to sizex.

In order to solve the problem, we distribute the data into the thread in the following code section:

```
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++) {
        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;
```

Figure 8: code to descompose

This decomposition consists in dividing the whole data structure in *howmany* blocks (4 in this case). Each thread is assigned with one of these blocks of (sizex*sizey)/howmany elements. Figure 7 shows graphically how those blocks are assigned and as we can see, the distribution is done from top to bottom and from left to right since variables i_start and i_end are initialized for each block depending on their blockid.

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.

In the relax_jacobi function in solver-omp.c we added **this** to the code:

```
int howmany=4;
#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;
```

Figure 9: code parallelization for jacobi

We wanted the first *for* to get a parallel execution while preserving diff variable to prevent overwritings and adding the *reduction(+:sum)* to accumulate the sum value after the execution of *sum += diff * diff;* by each thread.

In this jacobi version, a copy of the matrix is done after the call to *relax_jacobi* function:

case 0: // JACOBI
residual = relax_jacobi(param.u, param.uhelp, np, np);
// Copy uhelp into u
copy_mat(param.uhelp, param.u, np, np);
Break;

Since *relax_jacobi* is now parallelized but *copy_mat* isn't there will occur a bottleneck here so we also parallelized the copy_mat function:

```
#pragma omp parallel for
for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
```

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.

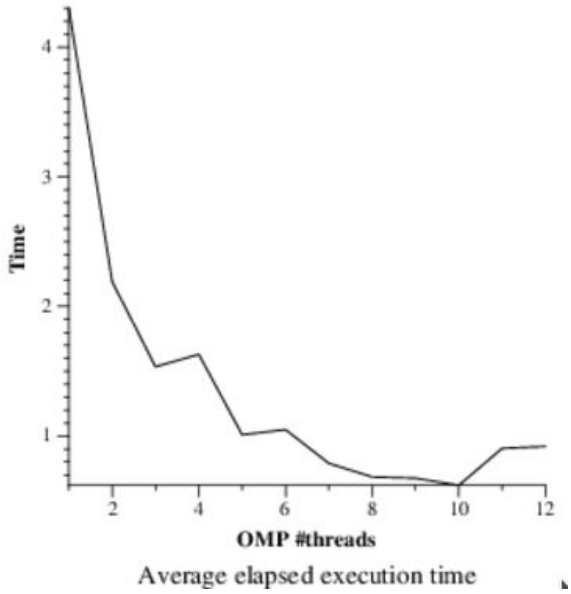


Figure 10: elapsed execution time program using jacobi

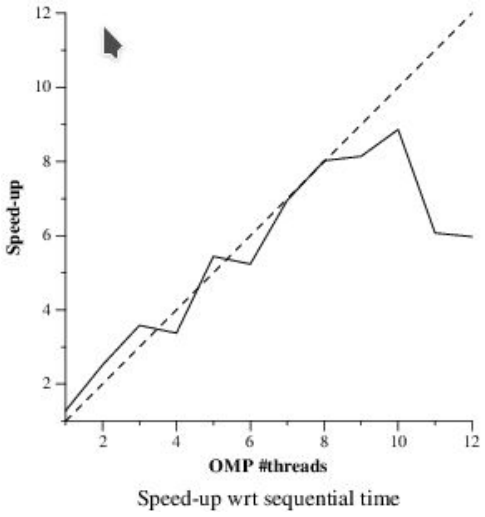


Figure 11: speedup obtained using jacobi

We can observe in figures 10 and 11 that speed-up and elapsed time plots indicate a decrement in the performance. This is due mainly because of the overheads created with every new task

OpenMP parallelization and execution analysis: Gauss-Seidel

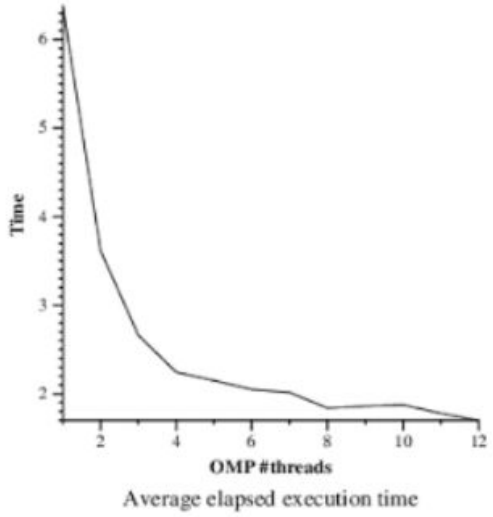
1. Include the relevant portions of the parallel code that implements the Gauss-Seidel solver, commenting how you implemented the synchronization between threads.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    int howmany = omp_get_max_threads();
    int processedBlocks[howmany];
    for(int i = 0; i < howmany; ++i) processedBlocks[i] = 0;
    int nBlocs = 8;


    #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;
}
```

Figure 12: Parallelized code for Gauss-Seidel

This parallelization has been more complex than the Jacobi one. Here, we divided the matrix to give equally rows to all threads and then divide the matrix space of each thread into blocks. We also made a processedBlocks vector of the size of num_threads. Using this strategy we assume that the dependencies are from top to bottom and from left to right.

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.



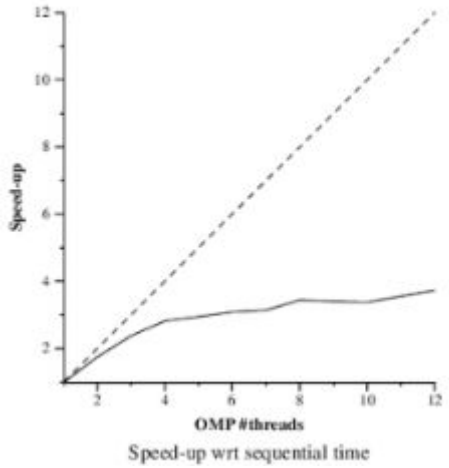


Figure 13: elapsed execution time for Gauss-Seidel

Figure 14: speed-up plot for Gauss-Seidel

Here in these plots we observe that the more threads are used the fastest the execution is but the improvements become less effective after 4 threads. In the speed-up plot we see something similar to what we see in the elapsed execution time plot as the speed up grows fast unit it gets to 4 threads and from there the growth is a bit less substantial.

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

To obtain the optimum value for the ratio computation/synchronization we submitted the program to multiple executions with 2, 4, 8, 16, 32, 64, and 128 blocks. Our results say that the optimal value is somewhere between 8 and 32 blocks, as blocks values of less than 8 blocks aren't enough optimized and more with than 64 blocks there is a lot of overhead due to fork/joins.