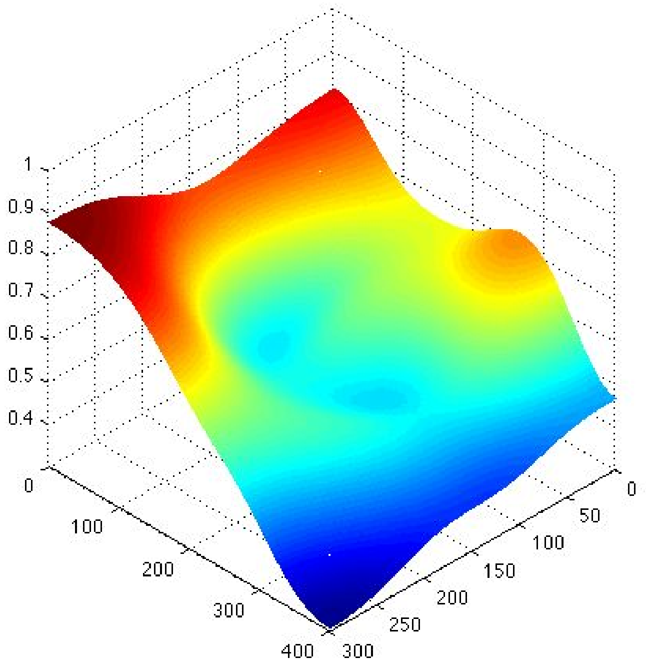
Laboratory 3

PAR-FIB

Group 22/05

QT 15-16



Albert Suàrez Molgó
Víctor Pérez Martos

INDEX

Analysis with Tareador

1.

JACOBI VERSION

```
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("InnerJacobi");
             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("InnerJacobi");
        }
      }
    }

    return sum;
}
```

Code 1: *relax_jacobi* function with tareador clauses.

Code 1 shows the code we have added to *solver-tareador.c*, we have created a tareador task in the most inner loop and disabled the sum variable because it caused dependency. In OpenMP, we can solve this problem by adding a *reduction* clause over *sum* variable to protect it.

Figure 1 reflects in a clearly way the dependences of the program over *sum* variable and the parallelism we accomplish by avoiding this dependence, this avoidance is reflected in the code with the *tareador_disable_object(&sum)* clause.

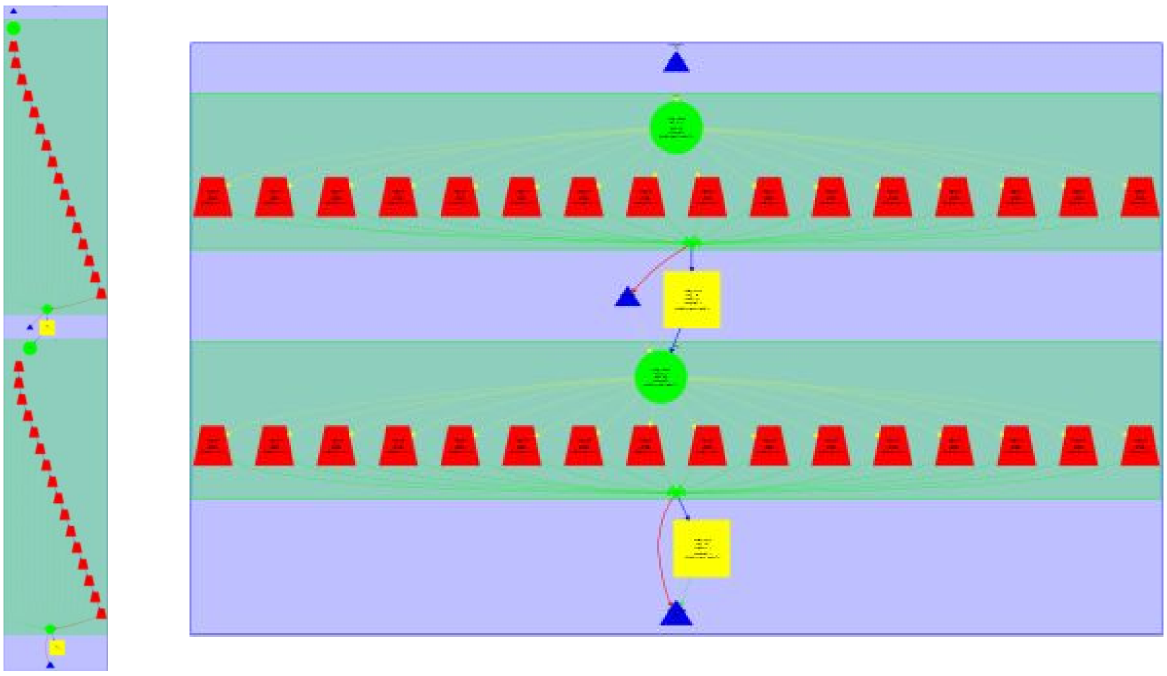


Figure 1: Dependency graphs of *jacobi* version without disable and disabling sum variable.

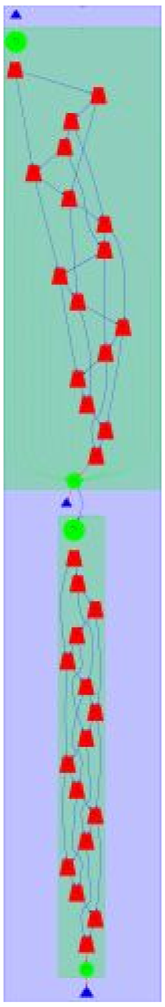
GAUSS-SEIDEL VERSION

```
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("InnerGauss");
            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("InnerGauss");
        }
      }
    }

    return sum;
}
```

Code 2: *relax_gauss* function with tareador clauses.



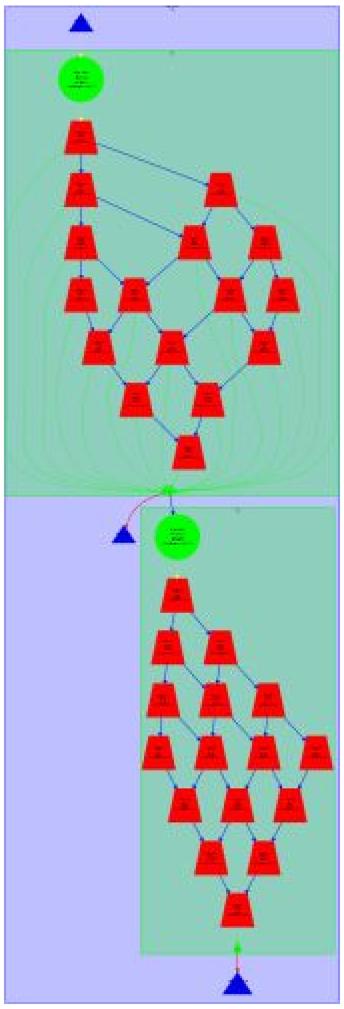


Figure 2: Dependency graphs of *gauss* version without disable and disabling sum variable.

Code 2 shows the code we have added to *solver-tareador.c*, we have created a tareador task in the most inner loop and disabled the sum variable because it caused dependency. In OpenMP, we can solve this problem by adding a *reduction* clause over *sum* variable to protect it.

On *Figure 2* we can see the graphs obtained with *tareador* without disable and disabling *sum* variable. Looking at both images, we can confirm that *sum* variable causes dependency between tasks and that we can accomplish a better performance by avoiding this dependency. Comparing *Figure 2* and *Figure 1* we can say that *Jacobi* version allows more parallelization than *Gauss-Seidel.*

OpenMP parallelization and execution analysis: Jacobi

1.

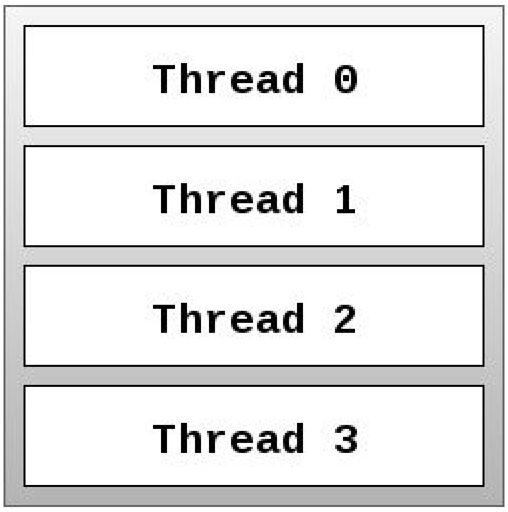


Figure 3: Data decomposition strategy

Supposing a table with *N* rows, we divide *N* between the number of processors, in this case 4 threads. So this way, thread 0 executes the first *N/4* rows, thread 1 the next *N/4* rows and so on with the other threads.

2.

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

Code 3: *relax_jacobi* function with omp clauses.

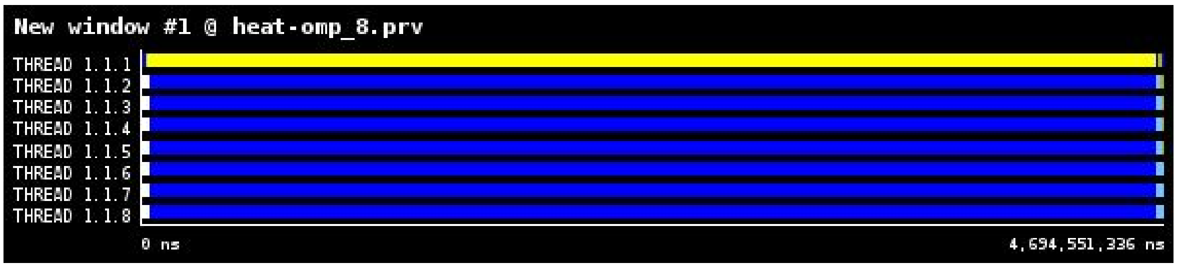


Figure 4: Paraver's trace of jacobi's execution.

On *code 3* we can see the parallelization of *relax-jacobi*. As we have mentioned on *exercise 1* from *Analysis with tareador,* we use a reduction clause to avoid the dependence caused by sum variable. Moreover, we use a *private* clause over diff variable

On *figure 4* we can see the *paraver* trace generated by the execution of the program.

3.

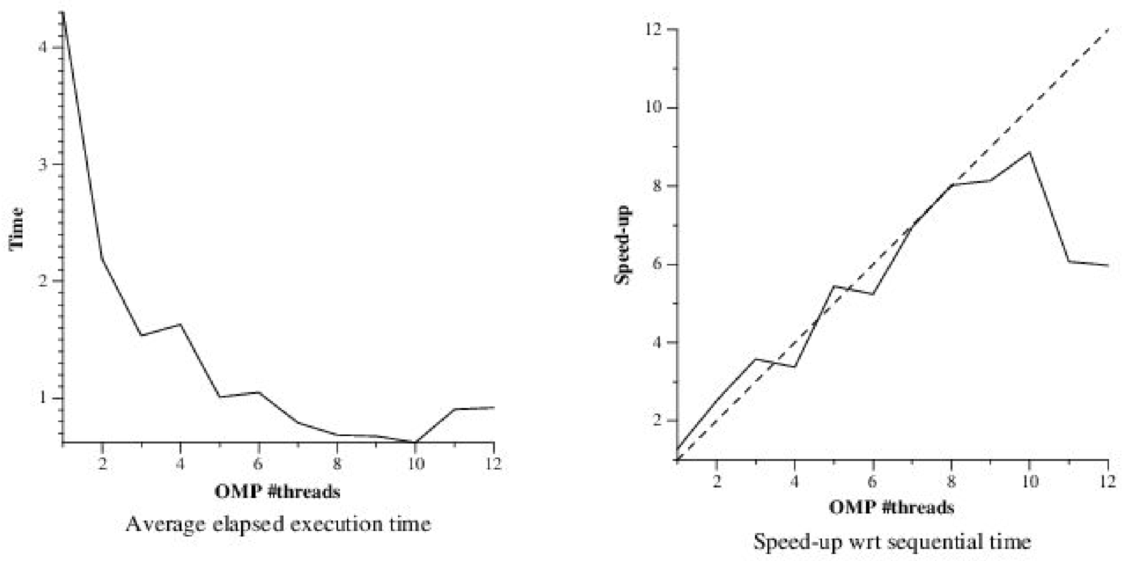


Figure 5: Plots of jacobi's execution with different number of threads.

Figure 5 shows the speedup obtained in different executions varying the number of threads. We can appreciate that the speedup obtained is bigger when we use more threads until we reach the amount of 10 threads, where it starts to decrease. This is caused due to the overhead generated by synchronization problems.

OpenMP parallelization and execution analysis: Gauss-Seidel

1.

```
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;
}
```

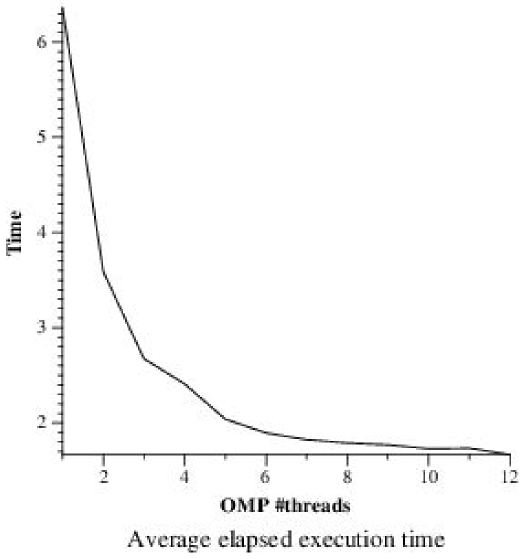
Code 4: *Gauss-Seidel* function with omp clauses.

Code 4 shows the parallelization of *Gauss-Seidel*, which is more complex than *relax-jacobi.*

For this algorithm, we have to divide the matrix to give equally rows to all the threads. Then, we have to divide the matrix space of each thread into blocks. With this decomposition, we assume that the dependence left to right is correct. However, we have to secure also the up dependence.

In order to do this, we have created the processedBlocks vector of the size of num_threads. We have done it with a pooling strategy of pooling and we wait the dependences needed, doing flush.

2.



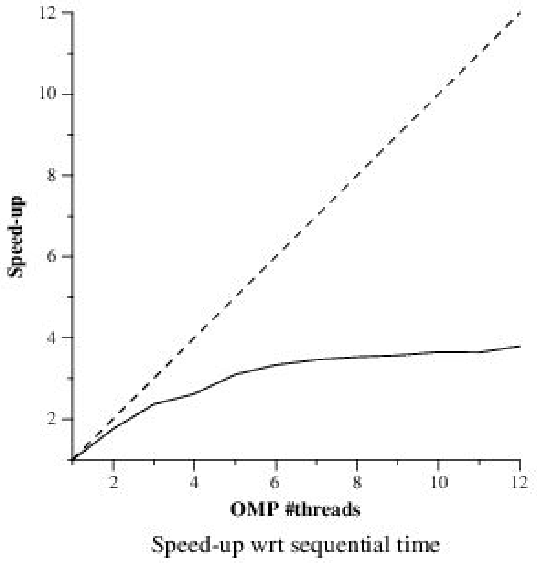


Figure 6: Plots of Gauss-Seidel execution with different number of threads.

Figure 6 shows the speedup obtained on *Gauss-Seidel* in different executions varying the number of threads. We can appreciate that there's i not much gain after using 4 threads and that the maximum speedup value obtained is around 4.

On *figure 7* we can see the *paraver* trace generated by the execution of the program.

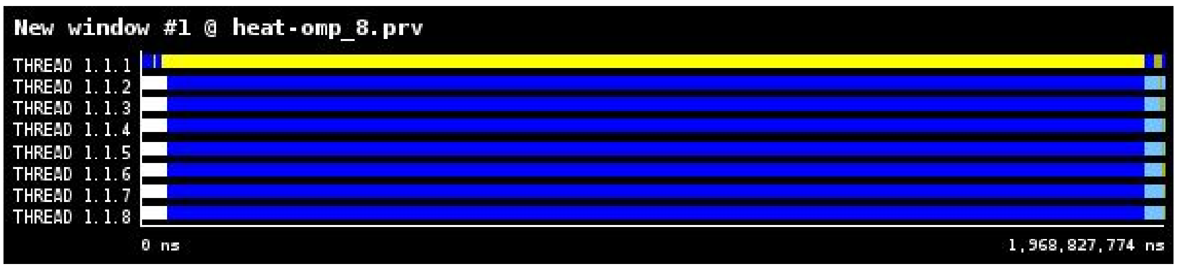


Figure 7: Paraver's trace of jacobi's execution.

3.

Looking at the code of Gauss, we can see that one of the most important things to do is to find the point with the optimum ratio between computation and synchronization. We do that on the basis that the execution is completed by 8 threads.

After a succession of executions with 2, 4, 8, 16, 32, 64 and 128 blocks, we have decided, comparing the results obtained, that the optimal value is between 8 and 32 blocks.

The main reason of that is because it has the smaller execution time. With small values there is a lot of computation per thread and with large values there is a lot of synchronization time.

Optional

1.

We have analyzed the code and all the possible solutions and we have decided that the best option is to create a task for each block with the *#pragma omp task* clause with an input dependences of the upper block and the left one, and an output dependences with the address of the block to compute.

But, when we want to recreate the previous reduction to *sum* variable, we can produce a false sharing problem. So, we should declare an array with as many lines as threads we have available and each line will have a size equal to the cache line size.

double partial_sum[NUM_THREADS][CACHE_LINE_SIZE/sizeof(int)]

So, each thread will increase its value in the first position of each line.

partial_sum[THREAD_ID][0] += diff*diff

Finally, once all tasks have finished their work, we will add all partial *sum* values in the global variable *sum* and we will return it.