PARAL·LELISME

Geometric (data) decomposition using implicit tasks: heat diffusion equation

Sergio Utrero Preciado (**par1122**)
Jordi Bru Carci (**par1104**)
21/05/2022
2021-2022.Q2

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

TABLE OF CONTENTS

Introduction

In this laboratory assignment we will learn and understand how to correctly parallelize a simulation of heat diffusion in a solid body (2D) with a geometric decomposition in depth.

In order to do this, we proceed to develop what the heat equation is. This equation computes how heat will be transferred over a 2D solid body. To explore how it works, we will use 2 different versions, the Jacobi equation and Gauss-Seidel equation. Both equations use the same solver code but at the end, there is only one difference. The Gauss-Seidel equation generates dependencies on how it iterates over the matrix (column based). On the other hand, the Jacobi equation iterates the matrix by rows.

```
int main( int argc, char *argv[] ) {
    //code...
    while(1) {
        switch( param.algorithm ) {
            case 0: // JACOBI
                residual = solve(param.u, param.uhelp, np, np);
                    // Copy uhelp into u
                    copy_mat(param.uhelp, param.u, np, np);
                    break;
            case 1: // GAUSS-SEIDEL
                    residual = solve(param.u, param.u, np, np);
                    break;
        default: // WRONG OPTION
            fprintf(stdout, "Error: solver not implemented, exiting execution \n");
            return EXIT_FAILURE;
            }

        iter++;

        // solution good enough ?
        if (residual < param.residual) break;

        // max. iteration reached ? (no limit with maxiter=0)
        if (param.maxiter>0 && iter>=param.maxiter) break;
    }
    //code...
}
```

Figure 1. Screenshot of the relevant part of the shared main

Figure 1 is the main code of boths equations. As we can see, there is a switch to call the solver of the operation we want to execute. There are two cases: case 0 is Jacobi, which calls its function and also makes a copy of it to save the computation for future use. Also there is the case 1 which is Gauss-Seidel that only calls the

function because this method does not need to save the computation and just needs to iterate through the main one.

Sequential heat diffusion program and analysis with Tareador

We will compile the sequential version of the heat code and then we will execute it with both solvers to check both performances. First of all, we are going to execute the generated binary using the Jacobi solver and then we will do the same procedure but with the Gauss-Seidel solver.

```
par1122@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 4.552
Flops and Flops per second: (11.182 GFlop => 2456.51 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 2. Output generated by the heat-Jacobi execution

```
par1122@boada-1:~/lab5$ ./heat test.dat -a 1 -o heat-gauss.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 6.011
Flops and Flops per second: (8.806 GFlop => 1464.92 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 3. Output generated by the heat-Gauss-Seidel execution

If we display both executions, it will generate figure 4 and clearly, we can see that there is no big difference. Actually there is a minimal distinction on the top right side. After understanding how both operations work, we can say that the reason behind this is that the solver of the Jacobi computes values based on the previous iteration and the Gauss-Seidel copies the data when it ends the section where it was working, resulting in a worse but cheaper execution time (as we can see in figures 2 and 3).

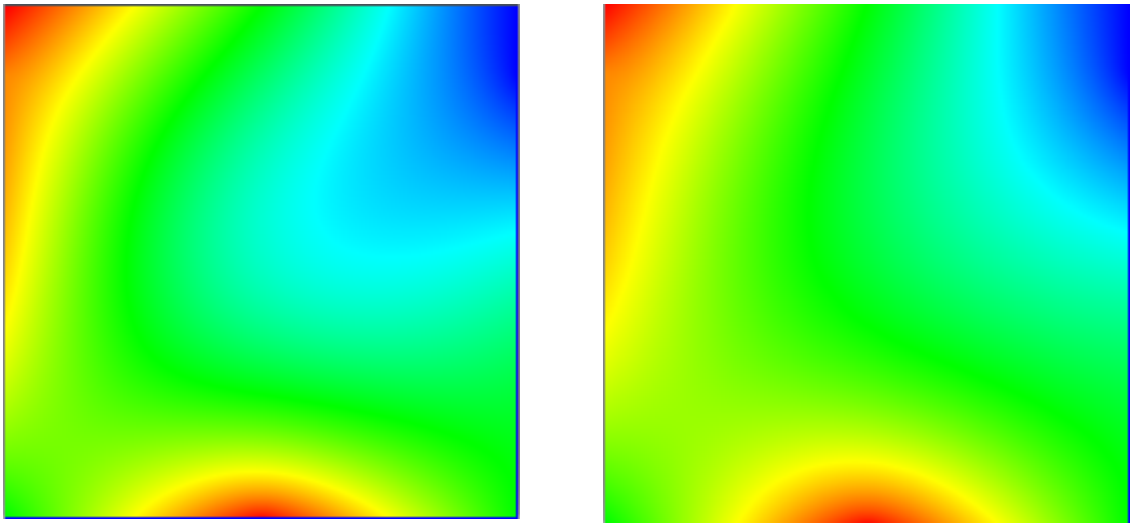

Figure 4. Image representing the temperature in each point of the 2D solid body with Jacobi (left) and Gauss-Seidel (right)

Analysis of task granularities and dependencies:

In this section we are going to analyze how code dependencies actually affect program parallelization. In order to do so, we will use the provided scripts to generate dependency graphs. But first of all we will declare tasks in the code as we will see in the following figure.

```
//code...
tareador_start_task("jacobi");
residual = solve(param.u, param.uhelp, np, np);
tareador_end_task("jacobi");
//code...

//code...
tareador_start_task("copy_mat");
        copy_mat(param.uhelp, param.u, np, np);
tareador_end_task("copy_mat");
//code...

//code...
tareador_start_task("gausseidel");
        residual = solve(param.u, param.u, np, np);
tareador_end_task("gausseidel");
//code...
```

Figure 5. Screenshot of the task declaration we are going to implement

To check that we have declared the tasks correctly, we will execute the run-tareador script with both operations to compare their respective dependency graphs (figure 6).

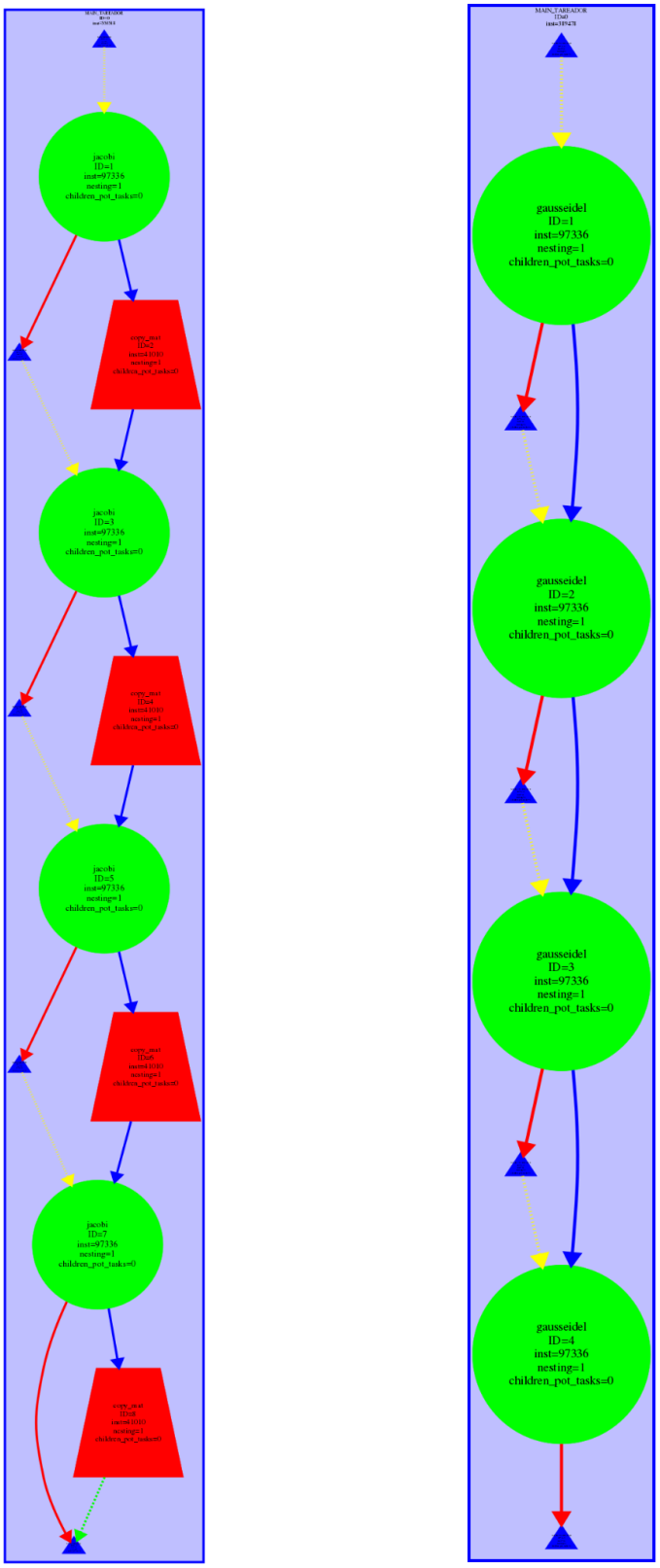

Figure 6. Dependency graphs generated of Jacobi (left) and Gauss-Seidel (right)

At first glance, we can say that with these task declarations no level of parallelism is obtained, at least not at this level of grain. Talking about the grain level, we can notice a slight difference between the two graphs in Figure 6. As we have already mentioned above, we can see that Jacobi calls the copy_mat function in addition to the solver call, but in contrast, Gauss-Seidel does it integrally with its solver.

To find a finer granularity we will take a close look at the solver-tareador.c code in order to find a good way to achieve our goal. After analyzing it, we have seen that the solver function splits the matrix into two blocks and then it computes a subset of rows and columns. What we are going to try is to declare one task for each block, as shown in figure 7.

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

    int nblocksi=4;
    int nblocksj=4;

    //tareador_disable_object(&sum);
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("solve_block");
        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++) {
                tmp = 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 = tmp - u[i*sizey+ j];
                sum += diff * diff;
                unew[i*sizey+j] = tmp;
          }
        }
        tareador_end_task("solve_block");
      }
    }
    //tareador_enable_object(&sum);

    return sum;
}
```

Figure 7. Screenshot of the task declaration per block on the solve function

Now we will proceed to execute our program with the new modifications to generate new dependency graphs in order to see if we have achieved our goal.

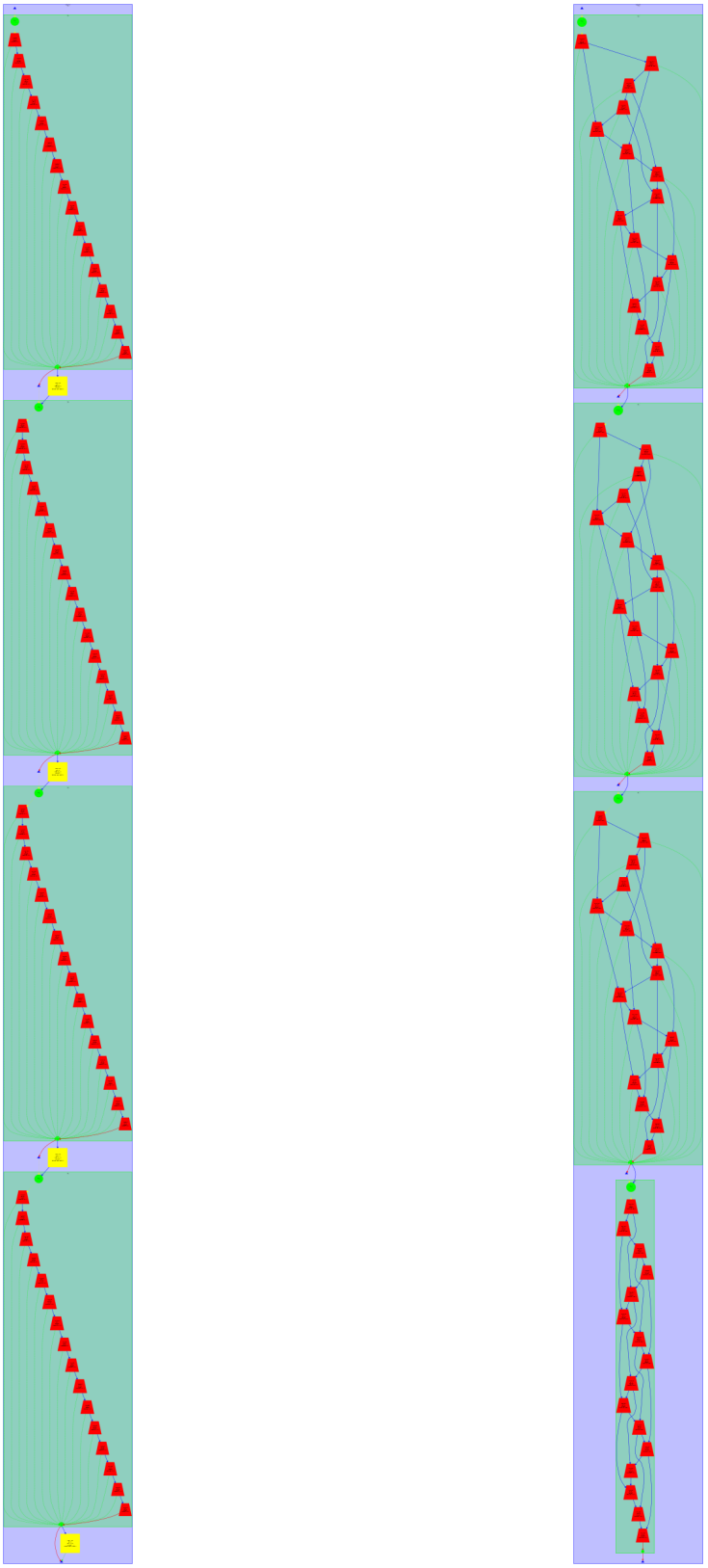

Figure 8. Dependency graph with task declaration at block level of Jacobi (left) and Gauss-Seidel (right).

As we can see in Figure 8, now the two methods, being block-based, have a much longer and more tangled graph structure. Still without achieving any level of parallelization. Everything follows a sequential execution.

Looking at the generated graphs and the code in Figure 7 we can assume that the variable that causes the serialization of the tasks is the variable sum since it would cause the expected bottleneck.

In order to emulate the effect of protecting the dependencies caused by this variable, we can uncomment the tareador_disable_object and tareador_enable_object_calls (same code as figure 7 but with the uncommented clauses). We proceed to re-execute our program with both methods so we can again see the dependency graphs generated with this new modification.

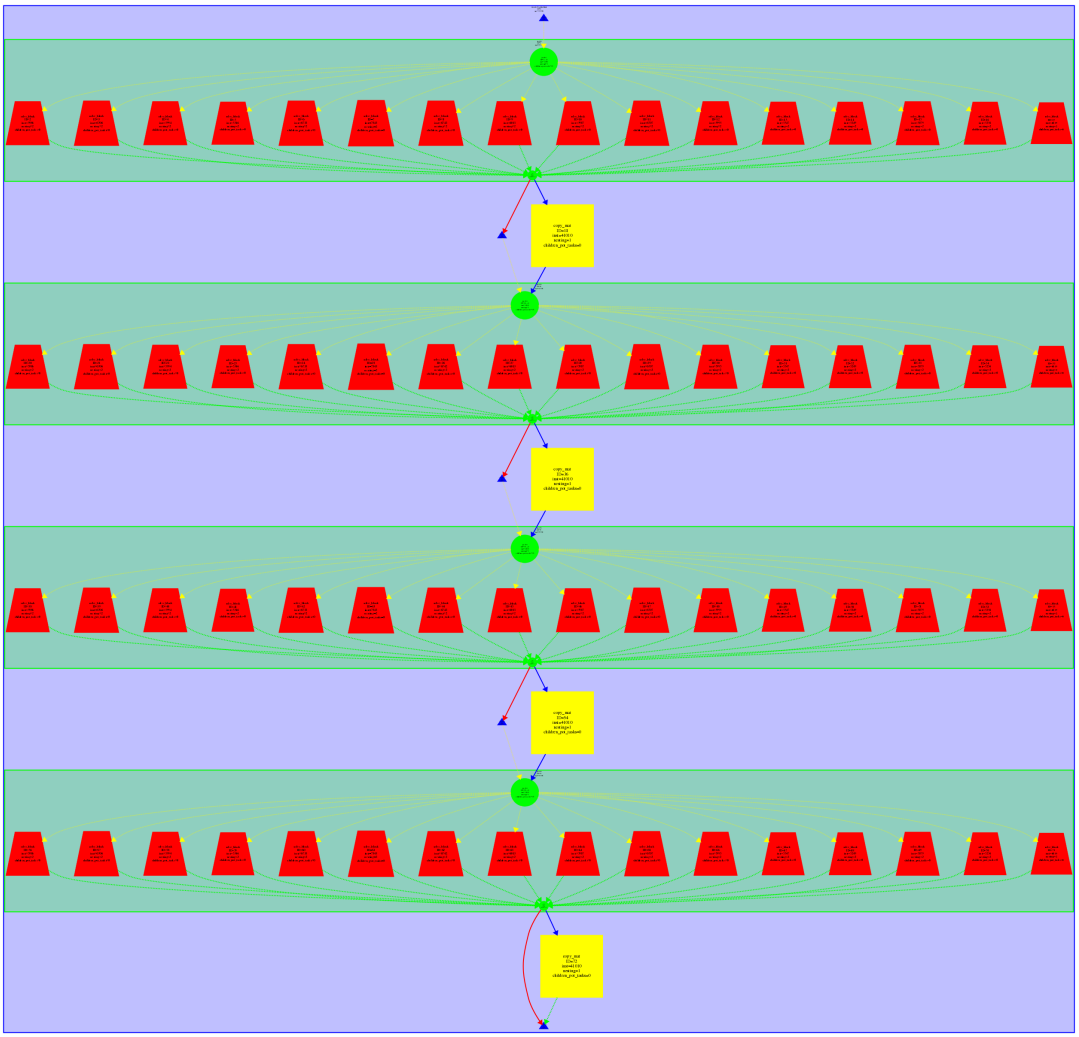

Figure 9. Dependency graph with task declaration at block level disabling sum on Jacobi

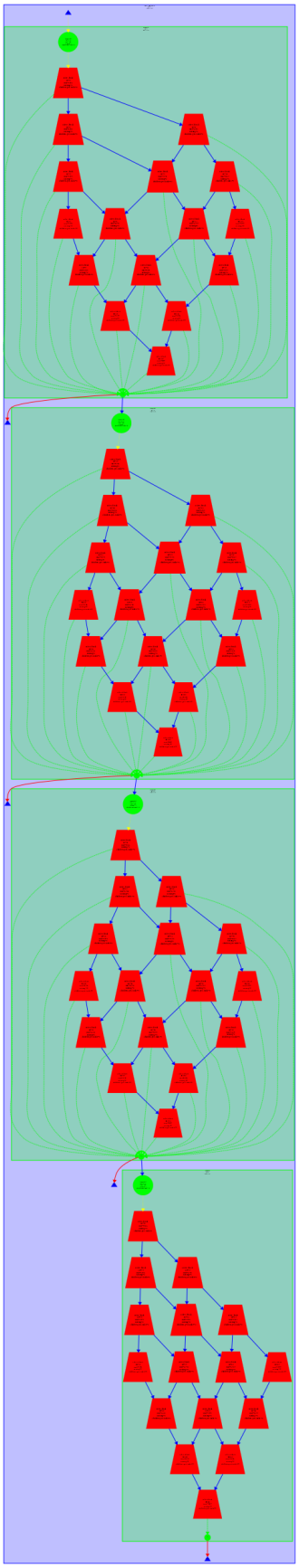

Figure 10. Dependency graph with task declaration at block level disabling sum on Gauss-Seidel

As shown in figure 9, the Jacobi solver has turned out to be embarrassingly parallel without dependencies between solve tasks nor solve_block ones. But we finally achieved a certain level of parallelism that we did not have before.

In the case of the Gauss solver (Figure 10) we can say that certain dependencies still remain. The same cannot be said of the previous one. We can see how for each call to the solve function, 16 solve_block tasks are generated as we saw in Figure 8. And in the same way, that each task depends on the block to its left and above. This is due, as we have already mentioned above, to the structure of the code.

If we ever wanted to protect the access of the sum variable with an OpenMP clause, we would use the reduction clause on sum so we will be able to avoid data races and incorrect solutions.

At the end of all, we can see how an improvement is achieved compared to figure 8. But to better understand its performance we will use the paraver tool and simulate the execution of each solver with 4 processors.

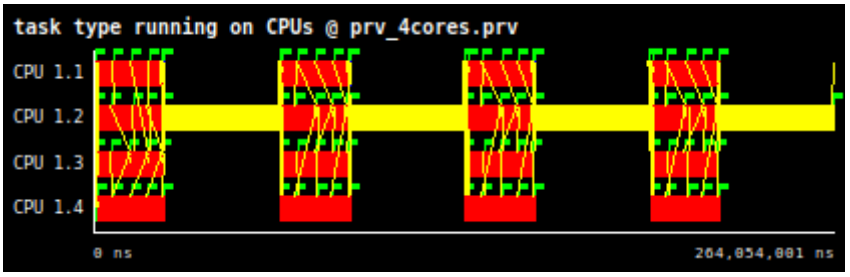

Figure 11. Chronogram of Jacobi method execution with 4 processors

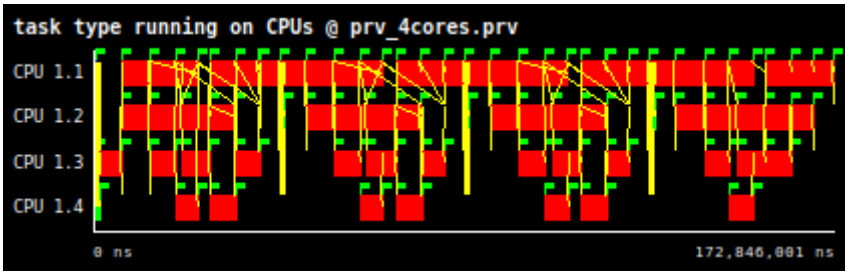

Figure 12. Chronogram of Jacobi method execution with 4 processors

We can easily see how Jacobi's method has a worse performance compared to Gauss-Seidel in terms of parallelization. Clearly there are big differences between the two figures (11 and 12). That is because we spend a lot of time executing the

copy_mat function which is executed sequentially. On the other hand, iIf we now look at the Gauss-Seidel method we can finally say that it does a good job of parallelizing the program and figure 12 demonstrates this.

In the previous paragraph we mentioned that the copy_mat function does not bring any improvement, in parallel speaking. Therefore, we are going to propose a possible modification to parallelize this function in order to improve the code. As we have done previously, we will declare the tasks by blocks (figure 13).

```
for (int blocki=0; blocki<nblocksi; ++blocki) {
  int i_start = lowerb(blocki, nblocksi, sizex);
  int i_end = upperb(blocki, nblocksi, sizex);
  for (int blockj=0; blockj<nblocksj; ++blockj) {
    int j_start = lowerb(blockj, nblocksj, sizey);
    int j_end = upperb(blockj, nblocksj, sizey);
    tareador_start_task("copy_mat_inside");
    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++)
        v[i*sizey+j] = u[i*sizey+j];
    tareador_end_task("copy_mat_inside");
  }
}
```

Figure 13. Screenshot of the task declaration per block on the copy_mat function

Now we are going to re-execute our code with the new modification in order to see how it affects the parallelization. If we inspect the dependencies we will see that we have reached a full embarrassingly parallel problem but this time in each of our functions (figure 14).

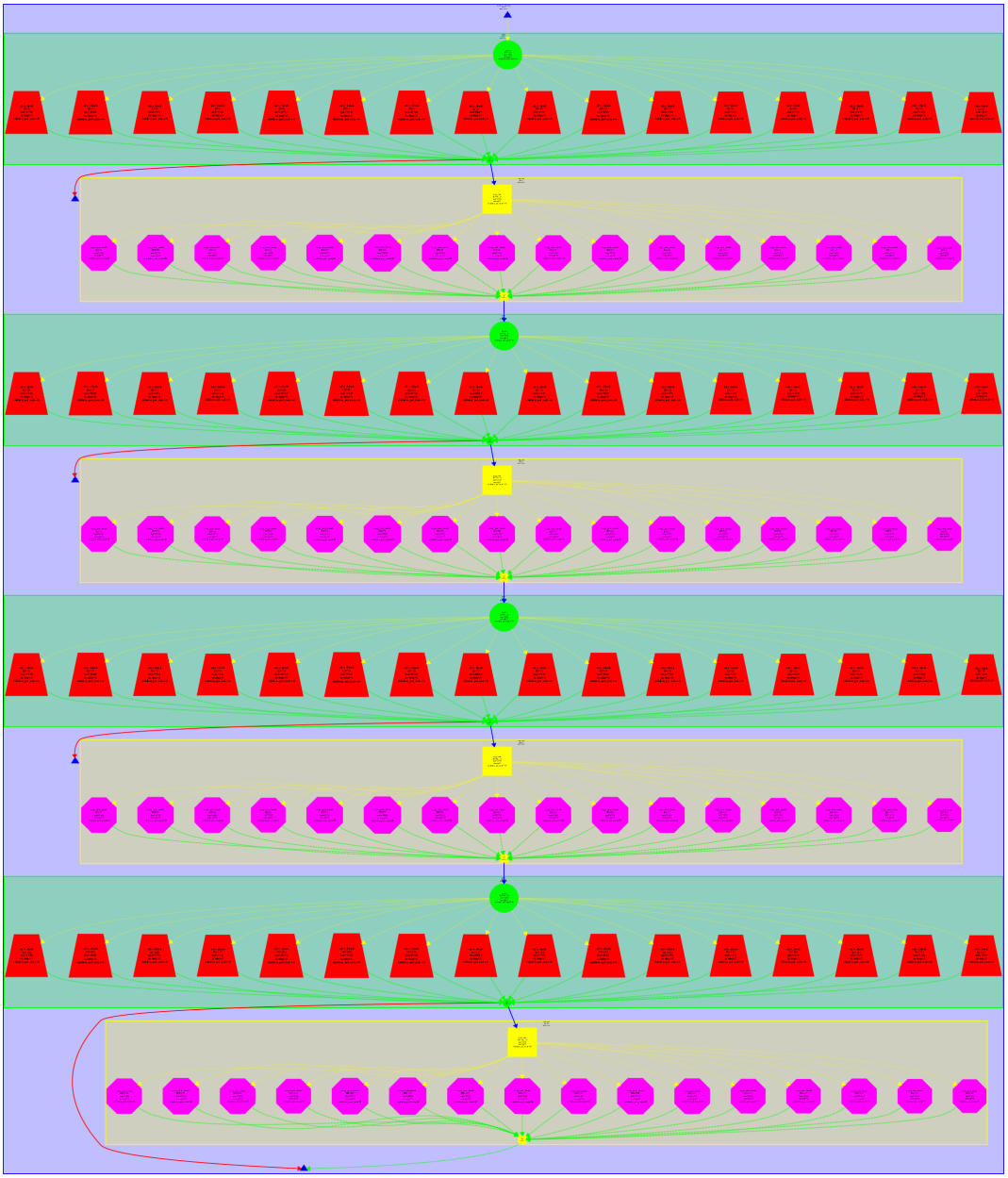

Figure 14. Dependency graph with task declaration at block level disabling sum and with and copy_mat improvmenet on Jacobi

Now we will simulate again the execution of our code with 4 processors with this last modification to see how it has finally improved.

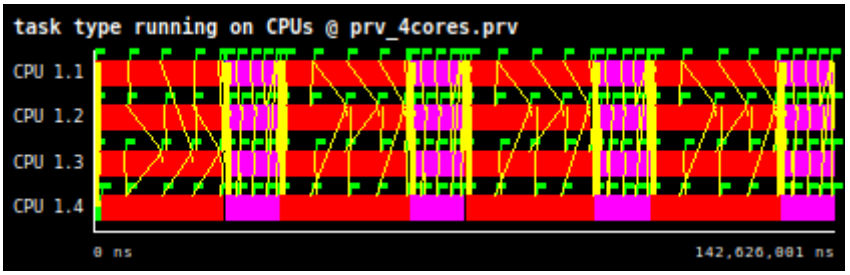

Figure 15. Chronogram of Jacobi method execution with 4 processors (after copy_mat parallelization)

Finally, we can see in Figure 15, how the parallelization and dependencies of the Jacobi method has improved a lot compared to the chronogram obtained in Figure 11.

Parallelisation of the heat equation solvers

In this lab session we will use the knowledge obtained in the previous sections to parallelize the two heat diffusion equations. This time, we will stop declaring explicit tasks and we will only work with implicit tasks. To do so, we will start with the Jacobi solver and then we will work with the Gauss-Seidel solver.

OpenMP parallelization and execution analysis: Jacobi

As mentioned before, we are going to parallelize our heat diffusion solver by dividing it into blocks. Specifically, we will divide the matrix as many times as the number of processors that are working on. In our case, we will execute it with 8 threads and then, at the end, we will be able to analyze it more deeply with modelfactors.

To finish implementing the started parallelization of the provided code, we have proceeded with the declaration of the parallel region.

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

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;

    #pragma omp parallel private(tmp, diff) reduction(+: sum)
    {
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        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++) {
                tmp = 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 = tmp - u[i*sizey+ j];
                sum += diff * diff;
                unew[i*sizey+j] = tmp;
          }
        }
      }
    }

    return sum;
}
```

Figure 16. Screenshot of the parallel region declaration on the solver-omp.c code

For this purpose, we have chosen to privatize the temporary variables tmp and diff. Also, we will implement the clause reduction into sum to avoid data races and incorrect solutions. We can see the modifications made in figure 16. We have also thought it appropriate to parallelize the copy_mat function but we will do it in a future section in order to see the evolution.

To check that we have done it correctly, we will proceed to execute our code along with the provided submit-omp.sh script with 8 processors. We can see the output generated in figure 17.

>sbatch ./submit-omp.sh heat-omp 0 8

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 1.766
Flops and Flops per second: (17.742 GFlop => 10044.81 MFlop/s)
Convergence to residual=118.416576: 25000 iterations
```

Figure 17. Output generated by the heat-Jacobi execution without the copy_mat parallelization

As we can see in Figure 17, the execution time has improved compared to our first execution of the jacobi solver. Thus, we can say that our parallelization is valid so we will keep moving forward by submitting the execution of the submit-strong-extrae.sh script to trace the execution for different numbers of processors (1, 2, 4, 8 and 12) and execute modelfactors.py.

>sbatch ./submit-strong-extrae.sh heat-omp 0

Number of processors	1	2	4	8	12
Elapsed time (sec)	4.89	2.87	1.88	1.49	1.60
Speedup	1.00	1.70	2.60	3.28	3.05
Efficiency	1.00	0.85	0.65	0.41	0.25

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8	12
Parallel fraction	81.01%				
Global efficiency	99.41%	98.98%	98.51%	88.05%	78.22%
-- Parallelization strategy efficiency	99.41%	98.03%	97.04%	96.96%	95.58%
-- Load balancing	100.00%	98.99%	99.63%	99.94%	99.90%
-- In execution efficiency	99.41%	99.03%	97.40%	97.01%	95.68%
-- Scalability for computation tasks	100.00%	100.97%	101.52%	90.81%	81.84%
-- IPC scalability	100.00%	101.05%	102.16%	99.67%	97.28%
-- Instruction scalability	100.00%	99.99%	99.97%	97.21%	94.31%
-- Frequency scalability	100.00%	99.93%	99.40%	93.73%	89.20%

Overheads in executing implicit tasks

Number of processors	1	2	4	8	12
Number of implicit tasks per thread (average)	1000.0	1000.0	1000.0	1000.0	1000.0
Useful duration for implicit tasks (average)	3937.31	1949.82	969.62	541.94	400.94
Load balancing for implicit tasks	1.0	0.99	1.0	1.0	1.0
Time in synchronization implicit tasks (average)	0	0	0	0	0
Time in fork/join implicit tasks (average)	23.3	59.02	33.26	16.79	18.51

Figure 18. Modelfactors output of the jacobi solver without copy_mat parallelization

Figure 18 shows in a more in-depth way the jacobi solver performance. We can clearly see , on the first table, that there is a big problem in terms of efficiency. The more processors are in use, the less efficiency appears.

But if we now examine the rest of the tables, we can say that all parameters maintain optimal levels and there is nothing out of place. We do not see any decrease in scalability but only a decrease in efficiency.

To see its performance, we have used the paraver tool to simulate traces with 8 processors. As we can see in figure 19, there are many pauses between peaks where we can suppose that they are the calls to the copy_mat function. Below the same figure we have put a capture of the same chronogram but zoomed-in. With this one, we can clearly see that we stay more than half of the time with just one thread computing copy_mat.

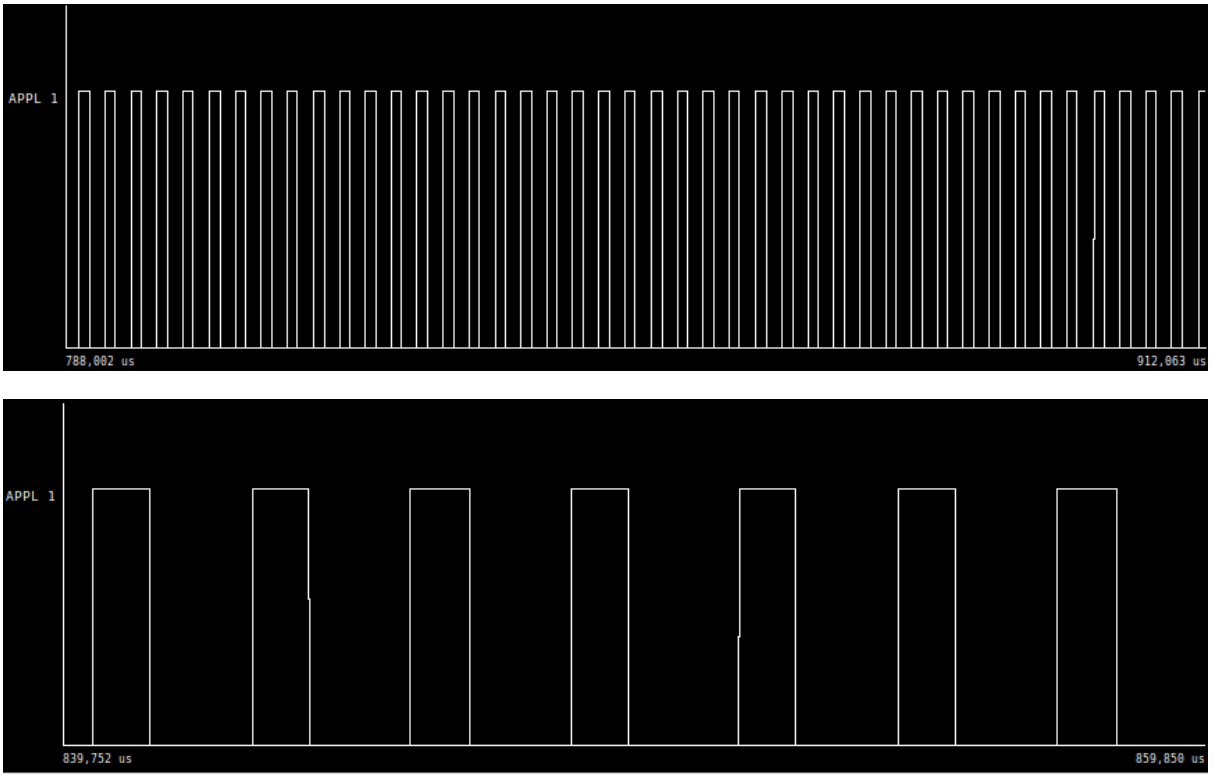

Figure 19. Chronograms of Jacobi solver with 8 processors (Immediate parallelism)

The solution we propose is clearly to parallelize the copy_mat function where we will declare another parallel region. Once we have our new version, to conclude talking about scalability, we will generate some plots so we can have more visual ways to compare data (see figures 24 and 25).

Now we proceed to parallelize the copy_mat function.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    #pragma omp parallel
    {
    //for (int blocki=0; blocki<nblocksi; ++blocki) {
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        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++)
            v[i*sizey+j] = u[i*sizey+j];
      }
    //}
  }
}
```

Figure 20. Screenshot of the parallel region declaration in the copy_mat function on the solver-omp.c code

In the copy_mat function we have commented out the lines that do not promote parallelism in order to create a robust parallel region.

Once the code is modified we proceed to follow the same steps that we have done previously in this same section. We will check its execution by submitting the code to boada with 8 threads.

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 0.725
Flops and Flops per second: (11.182 GFlop => 15414.49 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 21. Output generated by the heat-Jacobi execution with the copy_mat parallelization

As we can see in Figure 21, the execution time is reduced to a very good level. From a time of almost 2 seconds to no more than 1 second. So far, we can say that it has been a success. But before claiming victory we proceed to generate the modelfactors output from 1 to 12 processors to finally see if a real efficiency improvement is achieved.

Number of processors	1	2	4	8	12
Elapsed time (sec)	6.07	2.53	1.37	0.92	0.78
Speedup	1.00	2.40	4.43	6.60	7.78
Efficiency	1.00	1.20	1.11	0.83	0.65

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8	12
Parallel fraction	98.95%				
Global efficiency	99.00%	120.09%	112.84%	85.75%	68.31%
-- Parallelization strategy efficiency	99.00%	97.74%	96.65%	95.78%	94.95%
-- Load balancing	100.00%	98.88%	99.60%	99.48%	98.58%
-- In execution efficiency	99.00%	98.85%	97.04%	96.28%	96.31%
-- Scalability for computation tasks	100.00%	122.87%	116.75%	89.53%	71.95%
-- IPC scalability	100.00%	122.93%	117.45%	96.98%	81.04%
-- Instruction scalability	100.00%	99.98%	99.95%	97.81%	98.31%
-- Frequency scalability	100.00%	99.96%	99.45%	94.38%	90.31%

Overheads in executing implicit tasks

Number of processors	1	2	4	8	12
Number of implicit tasks per thread (average)	2000.0	2000.0	2000.0	2000.0	2000.0
Useful duration for implicit tasks (average)	2973.57	1210.05	636.75	415.16	344.41
Load balancing for implicit tasks	1.0	0.99	1.0	0.99	0.99
Time in synchronization implicit tasks (average)	0	0	0	0	0
Time in fork/join implicit tasks (average)	29.96	41.74	25.55	21.62	18.83

Figure 22. Modelfactors output of the jacobi solver with copy_mat parallelization

Looking at Figure 22, we can see that we have not only improved the speedup but also reduced the decrease in efficiency. As with the previous code modification, the rest of the parameters continue to maintain good levels. Actually, we can say that we have managed to improve our first version of the jacobi equation since we have parallelized it even more and with good levels. The copy_mat function plays an important role in the code since it is called every time we call the solver as well. That's why we really see an improvement.

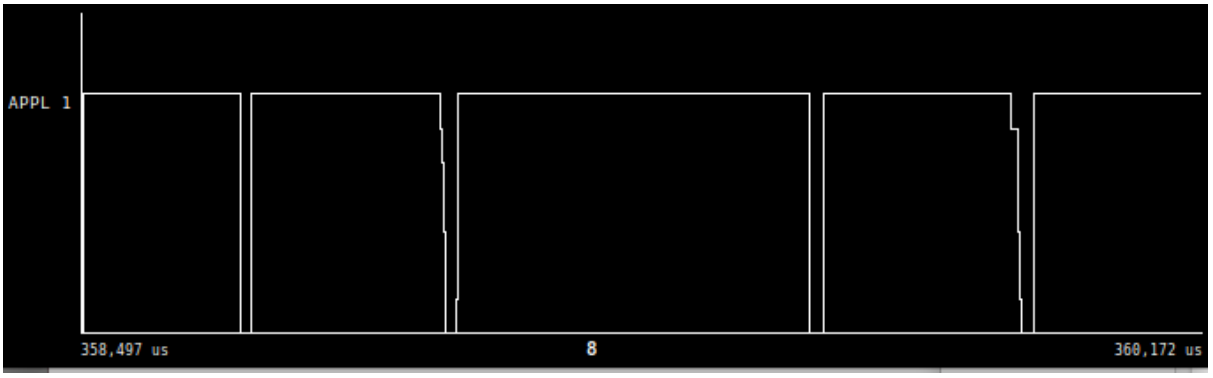

Figure 23. Chronograms of Jacobi solver with 8 processors with copy_mat parallelization (Immediate parallelism)

Before comparing the scalability of the two versions, we have found it appropriate to generate the same chronogram of the previous section (see figure 19) and thus be able to visually compare the levels of parallelism that we obtain.

This time we have captured the trace with a lot of zoom as we can see in the corners of the figure, since if we took a much larger range we could see perfectly a straight line. This line indicates how well parallelized our program is. In the previous version of our program we assumed that the gaps that appeared in the chronogram were the calls to the copy_mat function since we did not have it parallelized. In this case we can see how by staying at a very linear level, we can affirm that what generated those gaps was the said function.

To complete our analysis of our two parallelization versions of the jacobi solver, we will proceed to use submit-strong-omp.sh script to queue the execution of heat-omp and obtain the scalability plot of both versions.

> sbatch ./submit-strong-omp.sh 0

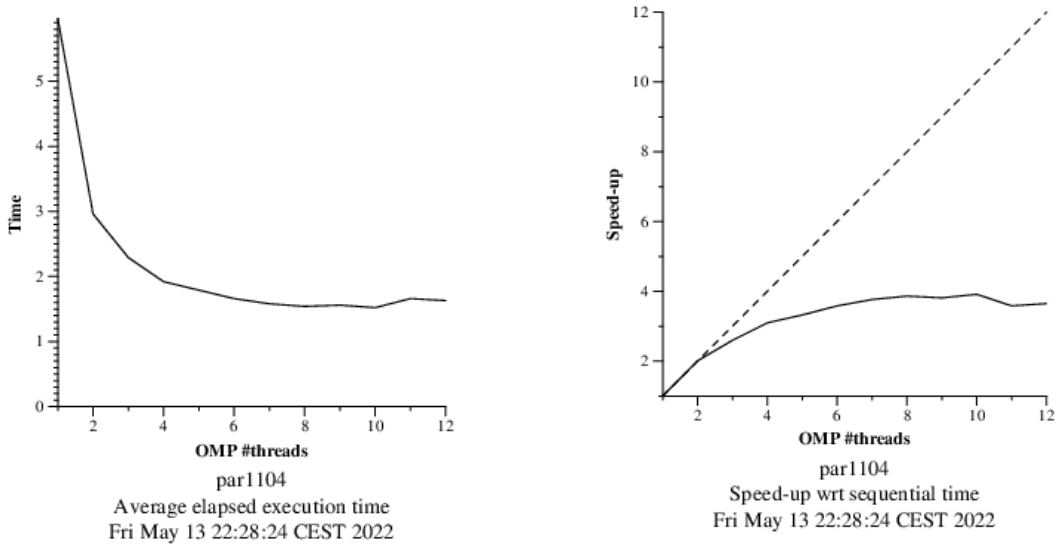

Figure 24. Scalability plots of Jacobi solver with 8 processors without copy_mat parallelization

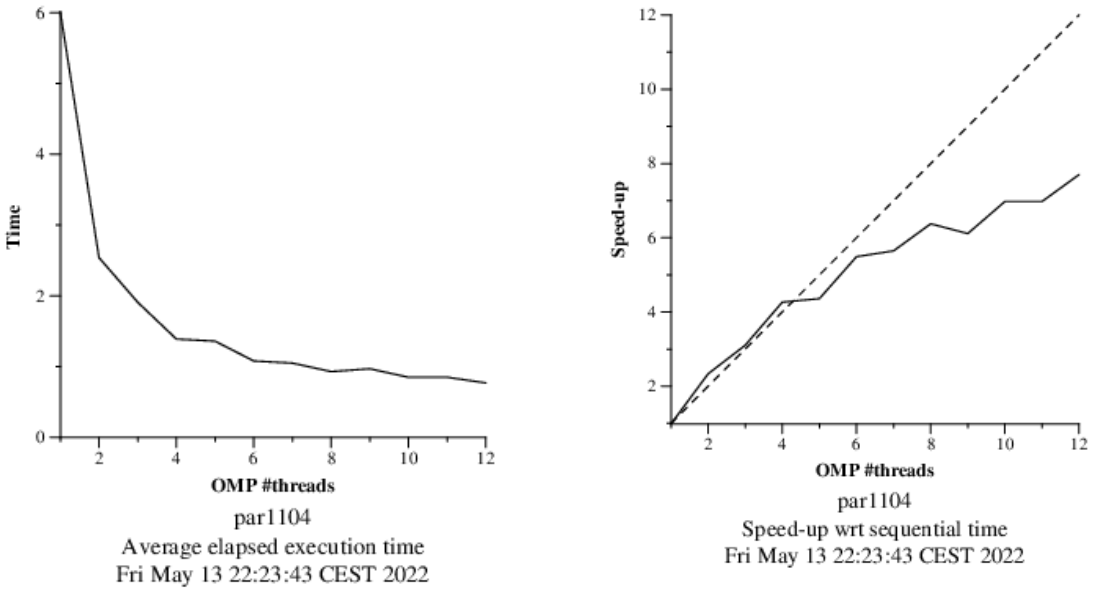

Figure 25. Scalability plots of Jacobi solver with 8 processors with copy_mat parallelization

If we compare the two scalability plots (see figures 24 and 25) we can see how the version with parallelized copy_mat obtains a better scalability to the point that we can say that it can scale to higher levels. So we could maintain the idea that it has a great strong scalability. It grows quite linearly, despite several peaks, and we can see how up to the first 4 threads it exceeds the ideal. From that level, its growth slows down but continues to improve steadily.

Observing also figure 24, we can say that this version needed to parallelize the copy_mat function, its growth was very flat although it was maintaining the level it had. These scalability plots just complement what we had mentioned before with the chronograms of both versions.

OpenMP parallelization and execution analysis: Gauss-Seidel

In order to carry out this part, it has been necessary to read "Annex 1: Creating your own synchronization objects", so our resolution will be based on the knowledge acquired in it.

For that, we will create an object for synchronized executions. In our case we will create a vector with nblocksi elements initialized at 0. We have also added an if statement that checks if it is a Jacobi or Gauss-Seidel execution and gets the thread to the correct lines of code (u == unew).

Whenever a thread has not the correct value on the control vector to get on with the computations it will stay in a do_while loop reading for that vector until the requirement is met.

For this synchronization vector to not create false sharing we have converted it to a matrix with a padding of fifteen integers to avoid two values to be in the same cache line. In this case we only use the element[n][0] of the matrix.

To protect those reads and writes of this shared matrix we have implemented an atomic read on the do_while loop and an atomic update on the read-write to update the value of the control element below.

```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;
    if(u == unew){
        int nblocksi=omp_get_max_threads();
        int next[nblocksi][16];
        int nblocksj=nblocksi;
        next[0][0] = nblocksj;
        for (int i = 1; i < nblocksi; i++) next[i][0] = 0;

    #pragma omp parallel private(tmp, diff) shared(next) reduction(+: sum)
    {
      int nexttmp;
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        do {
                #pragma omp atomic read
                nexttmp = next[blocki][0];
        } while (nexttmp < blockj + 1);
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        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++) {
                tmp = 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 = tmp - u[i*sizey+ j];
                sum += diff * diff;
                unew[i*sizey+j] = tmp;
          }
        }
        if (blocki < nblocksi-1) {
                #pragma omp atomic update
                next[blocki+1][0]++;
        }
      }
     }
    }
    else {
        //Jacobi
    }

     return sum;
}
```

Figure 26. Screenshot of the parallel region declaration on the solver-omp.c code

If we submit this code to Boada with eight threads we will get the following output:

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 1.663
Flops and Flops per second: (8.806 GFlop => 5296.67 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 27. Output generated by the heat-Gauss-Seidel execution

As we can see in the figure 27, the time is superior to Jacobi's version with copy_mat parallelization. We have a direct gain in time without the need of the copy_mat function because we update the matrix on each iteration.

To check for correctness we will apply a diff bash command between the sequential Gauss-Seidel heat diffusion result and this parallel .ppm. That output was correct.

To further explain the Gauss-Seidel version, we have used the powerful modelfactors tool to get an overview of how the code behaves. What stands out most is the low efficiency and scalability for computation tasks.

```
Overview of whole program execution metrics:
=============================================================================
   Number of processors |          1 |          2 |          4 |          8 |         12
=============================================================================
Elapsed time (sec)      |       7.95 |       6.00 |       3.53 |       2.06 |       1.53
Speedup                 |       1.00 |       1.33 |       2.25 |       3.86 |       5.19
Efficiency              |       1.00 |       0.66 |       0.56 |       0.48 |       0.43
=============================================================================

Overview of the Efficiency metrics in parallel fraction:
=============================================================================================================
                  Number of processors |          1 |          2 |          4 |          8 |         12
=============================================================================================================
Parallel fraction                      |     99.43% |
-------------------------------------------------------
Global efficiency                      |     99.82% |     66.31% |     56.53% |     49.04% |     44.34%
-- Parallelization strategy efficiency |     99.82% |     83.13% |     78.24% |     99.20% |     98.75%
   -- Load balancing                   |    100.00% |     83.31% |     78.52% |     99.95% |     99.97%
   -- In execution efficiency          |     99.82% |     99.79% |     99.64% |     99.25% |     98.78%
-- Scalability for computation tasks   |    100.00% |     79.77% |     72.25% |     49.43% |     44.90%
   -- IPC scalability                  |    100.00% |     93.90% |     91.68% |     99.79% |     99.11%
   -- Instruction scalability          |    100.00% |     84.95% |     78.78% |     53.33% |     51.10%
   -- Frequency scalability            |    100.00% |    100.00% |    100.03% |     92.88% |     88.65%
=============================================================================================================

Overheads in executing implicit tasks
=============================================================================================================================
                           Number of processors |               1 |               2 |               4 |               8 |              12
=============================================================================================================================
Number of implicit tasks per thread (average)    |          1000.0 |          1000.0 |          1000.0 |          1000.0 |          1000.0
Useful duration for implicit tasks (average)     |         7891.71 |          4946.7 |         2730.55 |         1995.67 |         1464.78
Load balancing for implicit tasks                |             1.0 |            0.83 |            0.79 |             1.0 |             1.0
Time in synchronization implicit tasks (average) |               0 |               0 |               0 |               0 |               0
Time in fork/join implicit tasks (average)       |           14.06 |         1995.27 |          1510.6 |           16.29 |           18.48
=============================================================================================================================
```

Figure 28. Modelfactors output of the Gauss-Seidel solver

As we can see in Figure 29, this code has a good parallelism as we have already seen in the second table of the previous figure, with a parallelisation strategy efficiency of almost 100%.

Figure 29. Chronograms of Gauss-Seidel solver with 8 processors (Immediate parallelism)

In figure 30 we can see how the scalability is worse with respect to Jacobi. This is what we expected from the previous modelfactors figure. The dependencies and the synchronization mechanisms restrain our threads in a way so that we cannot get to a linear speedup.

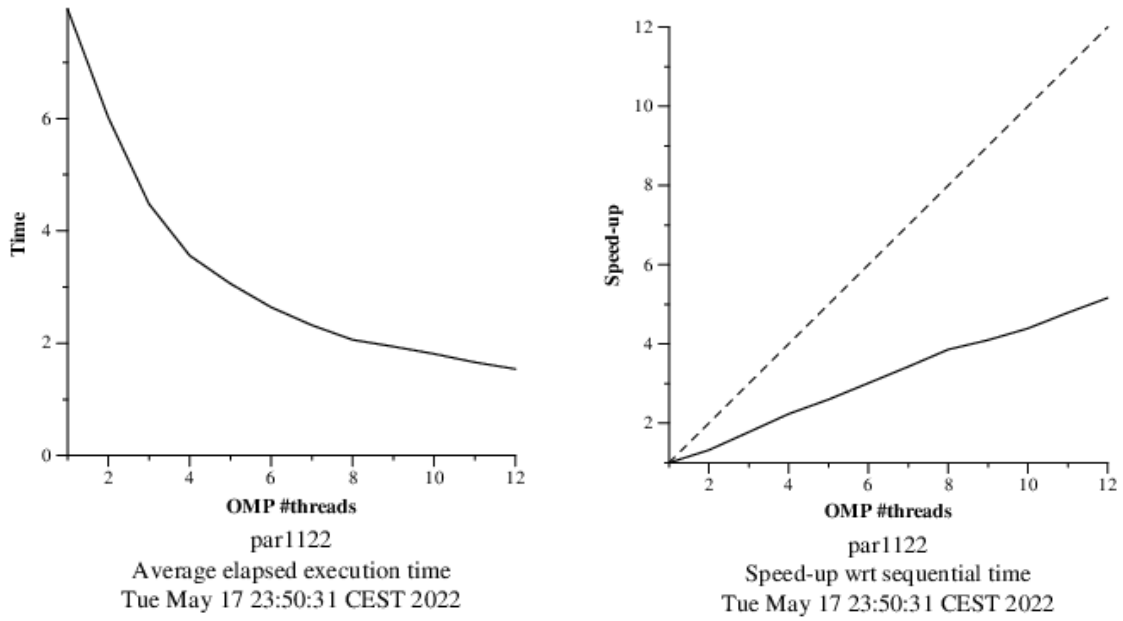

Figure 30. Scalability plots of Gauss-Seidel solver

For now we have been working with a grid of nblocksi x nblocks i, but we do not know if that kind of division is benefiting our execution times.

In order to exploit more parallelism in the execution of the solver, it has been proposed to us to modify the code so that we can change nblocksj.
We have chosen to implement it as nblocksj=userparam*nblocksi as we believe that with just a sweep of different continuous values we are able to check which ones are computationally optimal.

Figure 31 represents the execution times of the code in Figure 26 having made the above-mentioned change with different userparam with 2, 4, 8 and 12 threads. In this case we have modified the submit-userparam-omp.sh script to do an extensive sweep with nblocksi, 10 x nblocksi, 20 x nblocksi, up to 100 x nblocksi.
We have a time valley around the userparam 20, this means that it is optimal to divide our matrix into n rows and n*20 columns, so we will take that as the optimal value.

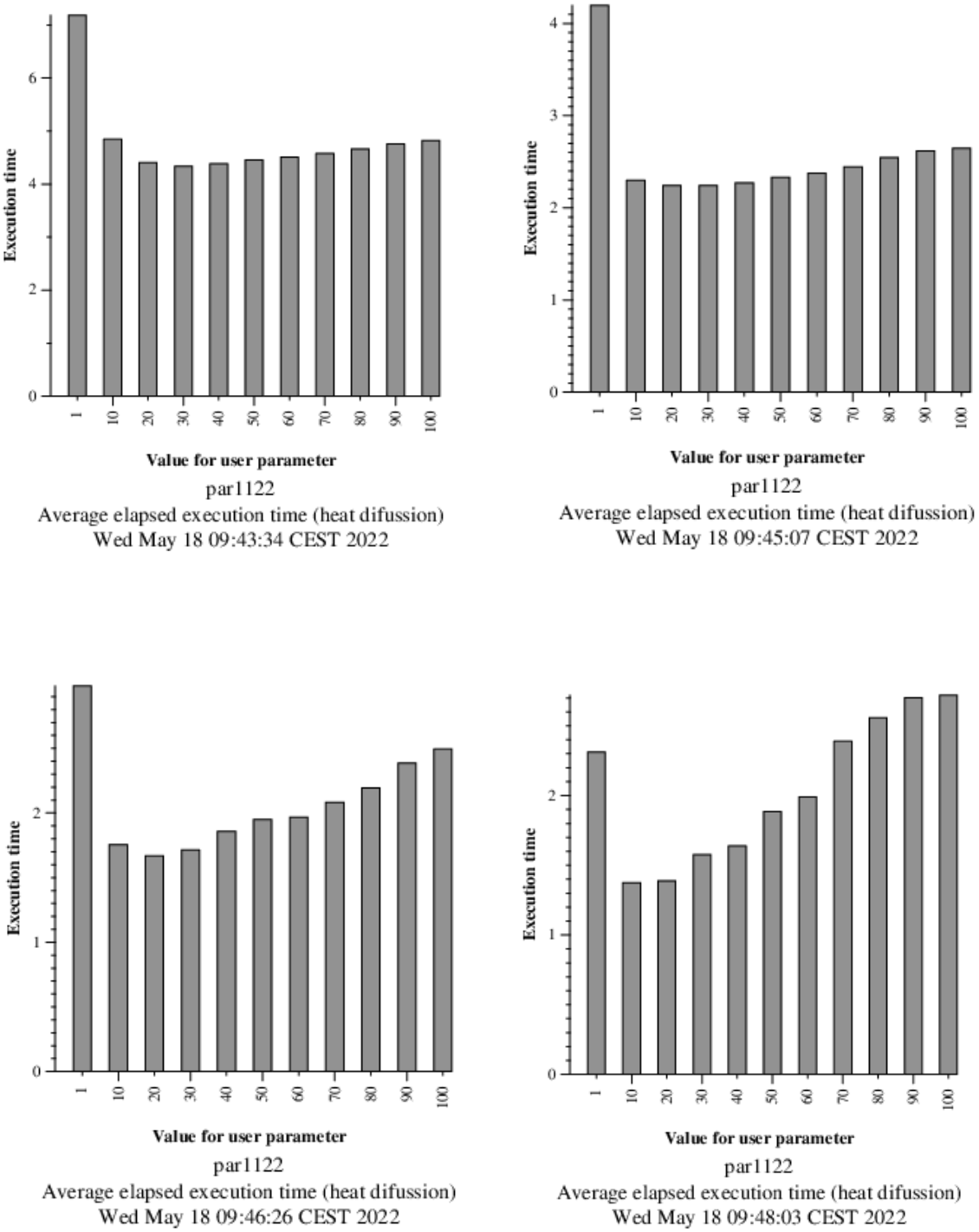

Figure 31. Execution times with different userparam with 2, 4, 8 and 12 threads

Optional 1

In this section we are going to implement the jacobi solver again but now we are going to declare explicit tasks and use task dependencies in order to compare the performance of both versions and understand the advantages and disadvantages of each one.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=omp_get_max_threads();
    int nblocksj=userparam*nblocksi;
    #pragma omp parallel
    #pragma omp single
    {
    #pragma omp taskloop
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int blocki = omp_get_thread_num();
      //same structure of copy_mat
      }
    }
  }
}

// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=omp_get_max_threads();
    int nblocksj=userparam*nblocksi;
    #pragma omp parallel
    #pragma omp single
    {
    #pragma omp taskloop private(tmp, diff) reduction(+: sum)
    //same structure of jacobi solver
     }
    }

    return sum;
}
```

Figure 32. Screenshot of the relevant part of this version of the solver-omp.c code

In Figure 32 we see how we have modified the code. We have commented out the body of both functions since the only changes to ensure what we wanted are made outside when declaring the tasks and their dependencies. But in this case we will assign a row of blocks on the i-axis for each thread instead of assigning one block per thread to exploit even better, as we have seen above, the parallelism.

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 2.527
Flops and Flops per second: (11.228 GFlop => 4443.15 MFlop/s)
Convergence to residual=0.000050: 15821 iterations
```

Figure 33. Generated output by the heat-Jacobi with explicit tasks execution

As we can see in Figure 33, the execution time of the jacobi solver declaring explicit tasks is worse than our previous version. To understand what causes a worse performance, we proceed to generate the modelfactors output.

```
Overview of whole program execution metrics:
=============================================================================================
   Number of processors |          1 |          2 |          4 |          8 |         12
=============================================================================================
Elapsed time (sec)      |       6.45 |       3.25 |       2.29 |       1.77 |       1.95
Speedup                 |       1.00 |       1.99 |       2.82 |       3.65 |       3.30
Efficiency              |       1.00 |       0.99 |       0.71 |       0.46 |       0.28
=============================================================================================

Overview of the Efficiency metrics in parallel fraction:
=============================================================================================================
                  Number of processors |          1 |          2 |          4 |          8 |          12
=============================================================================================================
Parallel fraction                      |     98.94% |
-------------------------------------------------------------
Global efficiency                      |     97.77% |     97.75% |     69.75% |     45.51% |     27.42%
-- Parallelization strategy efficiency |     97.77% |     95.51% |     86.58% |     81.13% |     70.51%
   -- Load balancing                   |    100.00% |     99.19% |     96.75% |     97.01% |     94.70%
   -- In execution efficiency          |     97.77% |     96.28% |     89.49% |     83.63% |     74.46%
-- Scalability for computation tasks   |    100.00% |    102.35% |     80.56% |     56.10% |     38.89%
   -- IPC scalability                  |    100.00% |    102.89% |     82.06% |     61.27% |     45.28%
   -- Instruction scalability          |    100.00% |     99.57% |     98.70% |     96.98% |     95.35%
   -- Frequency scalability            |    100.00% |     99.90% |     99.47% |     94.39% |     90.06%
=============================================================================================================

Statistics about explicit tasks in parallel fraction
=============================================================================================================================================
                          Number of processors |              1 |              2 |              4 |              8 |              12
=============================================================================================================================================
Number of explicit tasks executed (total)      |         2000.0 |         4000.0 |         8000.0 |        16000.0 |         24000.0
LB (number of explicit tasks executed)         |            1.0 |            1.0 |            1.0 |            1.0 |             1.0
LB (time executing explicit tasks)             |            1.0 |           0.99 |           0.97 |           0.97 |            0.95
Time per explicit task (average)               |        3101.41 |        1512.97 |         959.14 |         686.95 |          659.99
Overhead per explicit task (synch %)           |           0.44 |           3.46 |           13.8 |           20.9 |            39.5
Overhead per explicit task (sched %)           |           1.45 |           0.62 |            0.7 |           0.76 |            0.77
Number of taskwait/taskgroup (total)           |         3000.0 |         3000.0 |         3000.0 |         3000.0 |          3000.0
=============================================================================================================================================
```

Figure 33. Modelfactors output of the Jacobi solver with explicit tasks declaration

After generating the tables of figure 34, we have been able to verify that actually the performance of the jacobi solver declaring explicit tasks is much worse (see figure 22). In the first table we can already see at a glance how the speedup does not grow as much as the previous version, but the most worrying thing is that the efficiency decreases to a poor level. We can already see in the second table how the global efficiency drops to 27% when in our previous version it reached 78%.

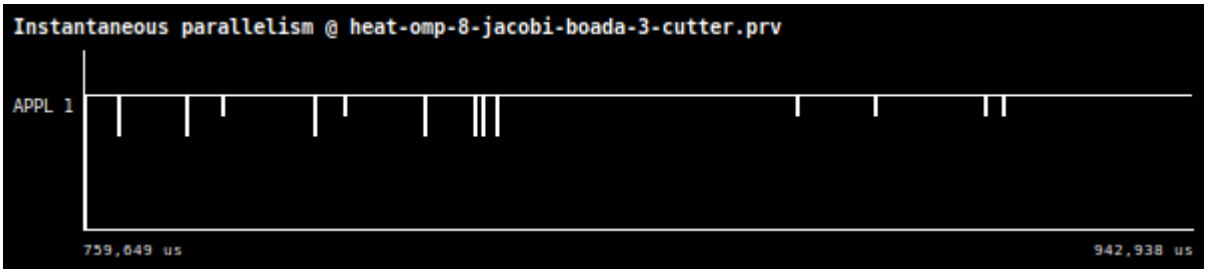

Figure 35. Immediate parallelism chronogram of Jacobi solver with 8 processors (explicit tasks declared)

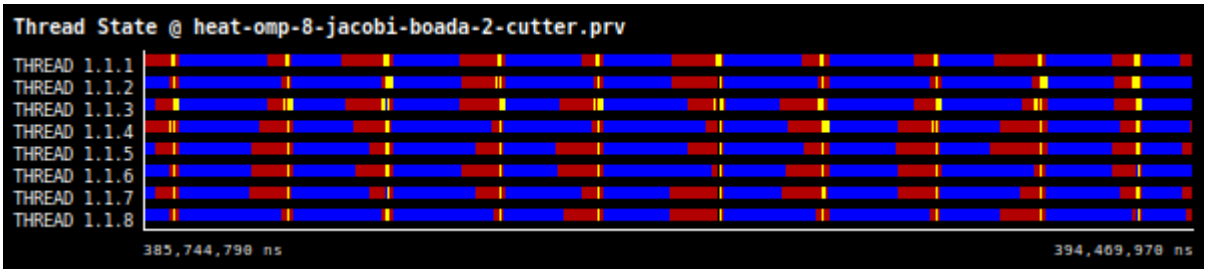

Figure 36. Thread state chronogram of Jacobi solver with 8 processors (explicit tasks declared)

We have generated the following traces to have a more visual view of the performance of this version. Regardless of the poor performance, we can see in Figure 35 how the parallelism is still at very good levels. If we look at figure 36, we can see how the threads spend a lot of time synchronizing between blocks (red color). Figure 37 helps us to understand the state of the threads in the execution seeing how they spend approximately 20% of their execution synchronizing causing a decrease in speedup and efficiency. This did not happen in the previous version since this computation was done by the implicit tasks having the functions privatized.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	75.35 %	22.68 %	1.98 %
THREAD 1.1.2	82.28 %	17.18 %	0.55 %
THREAD 1.1.3	78.02 %	20.77 %	1.21 %
THREAD 1.1.4	81.29 %	17.67 %	1.04 %
THREAD 1.1.5	77.07 %	22.46 %	0.48 %
THREAD 1.1.6	83.26 %	16.45 %	0.29 %
THREAD 1.1.7	76.88 %	22.74 %	0.38 %
THREAD 1.1.8	80.10 %	19.14 %	0.76 %

Figure 37. Histogram of the execution of jacobi solver with 8 processors (explicit tasks declared)

Optional 2

In this section we will perform the same analysis as in the "Optional 1" section but with the Gauss-Seidel solver. We will proceed to declare explicit tasks and use task dependencies to understand how the two versions differ in performance.

```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;
    if(u == unew){
        int nblocksi=omp_get_max_threads();
        int next[nblocksi][16];
        int nblocksj=userparam*nblocksi;
        next[0][0] = nblocksj;
        for (int i = 1; i < nblocksi; i++) next[i][0] = 0;
        #pragma omp parallel
        #pragma omp single
        {
      #pragma omp taskloop private(tmp, diff) shared(next) reduction(+: sum)
      for (int blocki = 0; blocki < nblocksi; blocki++) {
      // same structure of gauss-seidel solver
    }
   }
  }

    return sum;
}
```

Figure 38. Screenshot of the relevant part of this version of the solver-omp.c code

As we can see in Figure 39, the execution time of the Gauss-Seidel solver declaring explicit tasks is worse than our previous version. To understand what causes a worse performance, we proceed to generate the modelfactors output.

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 2.146
Flops and Flops per second: (8.806 GFlop => 4103.49 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 39. Generated output by the heat-Gauss-Seidel with explicit tasks execution

Figure 40 confirms that this new version of the code is worse than the one seen before, if we compare it with figure 28 we see that the speedup does not grow as much and the efficiency is lower.

Overview of whole program execution metrics:

Number of processors	1	2	4	8	12
Elapsed time (sec)	8.35	6.39	3.84	2.27	1.82
Speedup	1.00	1.31	2.17	3.67	4.60
Efficiency	1.00	0.65	0.54	0.46	0.38

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8	12
Parallel fraction	99.47%				
Global efficiency	99.53%	65.13%	54.41%	46.34%	38.99%
-- Parallelization strategy efficiency	99.53%	82.62%	77.25%	73.87%	71.85%
-- Load balancing	100.00%	93.77%	89.08%	93.20%	95.02%
-- In execution efficiency	99.53%	88.11%	86.72%	79.27%	75.62%
-- Scalability for computation tasks	100.00%	78.82%	70.44%	62.73%	54.27%
-- IPC scalability	100.00%	97.66%	96.64%	96.03%	93.25%
-- Instruction scalability	100.00%	80.85%	73.29%	68.91%	64.74%
-- Frequency scalability	100.00%	99.82%	99.44%	94.79%	89.88%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8	12
Number of explicit tasks executed (total)	1000.0	2000.0	4000.0	8000.0	12000.0
LB (number of explicit tasks executed)	1.0	1.0	1.0	1.0	1.0
LB (time executing explicit tasks)	1.0	0.94	0.89	0.93	0.95
Time per explicit task (average)	8248.18	5229.46	2923.41	1637.98	1259.97
Overhead per explicit task (synch %)	0.15	20.73	29.0	34.54	37.96
Overhead per explicit task (sched %)	0.24	0.17	0.2	0.29	0.37
Number of taskwait/taskgroup (total)	2000.0	2000.0	2000.0	2000.0	2000.0

Figure 40. Modelfactors output of the Gauss-Seidel solver with explicit tasks declaration

In the same way that we did in the Gauss-Seidel solver part, the figure of the immediate parallelism helps us to graphically see the quality of the parallelism of our code. In this case we see that it is much worse than in figure 29, so we can say that it does not improve.

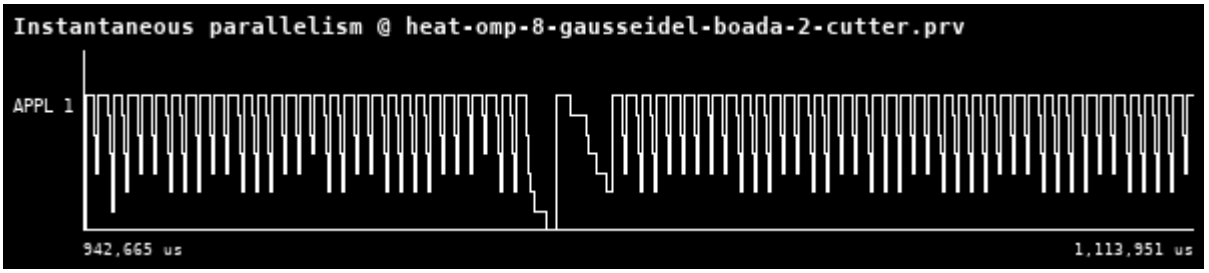

Figure 41. Immediate parallelism chronogram of Gauss-Seidel solver with 8 processors (explicit tasks declared)

With figure 42 we can see that the same thing happens as in the other solver. The threads spend a relatively long time synchronizing. For it we will generate the chronogram of figure 43 to see it in a more numerical way.

Figure 42. Thread state chronogram of Gauss-Seidel solver with 8 processors (explicit tasks declared)

Figure 43 shows the state of the threads in execution, showing how they spend approximately 25%-30% of their execution time synchronizing, thus decreasing speedup and efficiency.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	72.55 %	26.80 %	0.65 %
THREAD 1.1.2	73.15 %	26.67 %	0.18 %
THREAD 1.1.3	78.53 %	21.12 %	0.35 %
THREAD 1.1.4	79.70 %	19.90 %	0.39 %
THREAD 1.1.5	73.51 %	26.27 %	0.22 %
THREAD 1.1.6	70.91 %	28.99 %	0.10 %
THREAD 1.1.7	72.18 %	27.67 %	0.15 %
THREAD 1.1.8	73.81 %	26.00 %	0.19 %

Figure 43. Histogram of the execution of Gauss-Seidel solver with 8 processors (explicit tasks declared)

Final conclusions

In this assignment we have understood how difficult it is to make a program with good levels of parallelism, and we analyze it in two different ways. Using only implicit tasks and then using explicit tasks and dependencies between tasks as well. For this we have been modifying the initial codes of Jacobi and Gauss-Seidel solvers.

The most tedious part of making the mentioned modifications has been to check at all times the dependencies between tasks and also to try to achieve with each version, a better performance.

Once finished all the versions of the two solvers we can say that the Jacobi solver has been the fastest to implement a good level of parallelism using implicit tasks only. Then in the optional part 1 we have seen how good results can be achieved but nothing compared to our first version, since on the first one the execution time is improved a lot.

If we focus on the Gauss-Seidel solver, we can say almost the same. The version in which we only implement implicit tasks improves quite a lot of the performance if we compare it with the rest. Although really, the Jacobi-solver ends up having better results at first sight.

Talking about parallelism, all the versions achieve good levels, although the versions with explicit declared tasks have some peaks that imply a worse version.

As a general idea, the versions with implicit tasks have obtained better results and if we, as programmers, must choose which method to implement, in this case we would choose our first two versions of each solver. We must keep in mind that the method of parallelism that is implemented depends on what we are looking for in our program, and in this case we have it clear.

Final survey

modelfactors -> 10 because it helps a lot to see everything with numbers and it is easy to compare.

tareador -> essential because of the tdg. So it is a 10 for us.

extrae + paraver -> It is really a very useful tool but difficult to understand, especially at the beginning. It was really hard to get familiar with this new tool, but when you make it your own, it ends up being very useful to complement our explanations. We rate it as a 6,5.