PAR Laboratory Assignment
Lab 5: Geometric (data) decomposition using implicit tasks: heat diffusion equation

Natalia Dai
Daniel Ruiz Jiménez

Username: par2306
par2317
Date: 02/06/2022
Spring 2021-22

INDEX

1. Introduction

In the final laboratory session, we will focus on data decomposition using implicit tasks. Unlike the previous 2 laboratory exercises, which were focused on explicit tasking using iterative and recursive strategies respectively, this one will be centered around tasks whose performance is determined by the data they have to deal with. In our case, we will use 2 iterative methods to solve linear systems of algebraic equations: Jacobi and Gauss-Seidel solvers.

On the first day, we will simulate parallelism with Tareador to check how the 2 solvers behave. We will use a program that simulates how heat dissipates in a solid body. The main difference between the 2 systems is that one uses a temporary data structure to save the data that we are operating with while the other manipulates the data structure itself.

On the second day, we will parallelise both heat equation solvers with our regular OpenMP clauses, to check if their performance increases or decreases.

2. Sequential heat diffusion program and analysis with Tareador

We compiled and executed the program heat.c with both Jacobi and Gauss-Seidel solvers, and with "display" we visualised the image files in .ppm:

Jacobi:

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: **4.660**
Flops and Flops per second: (11.182 GFlop => 2399.36 MFlop/s)
Convergence to residual=0.000050: 15756 iterations

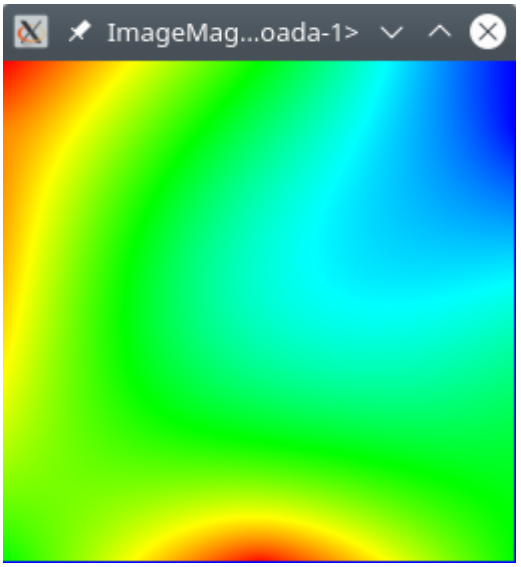


Figure 1: display heat-jacobi.ppm

Gauss-Seidel:

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: **6.035**
Flops and Flops per second: (8.806 GFlop => 1459.22 MFlop/s)
Convergence to residual=0.000050: 12409 iterations

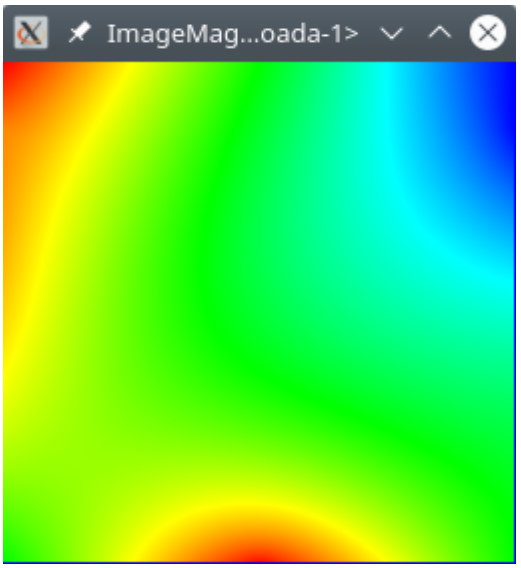


Figure 2: display heat-gauss.ppm

As we can see, both figures are very similar but in comparison, there are red and blue parts that are slightly different. We will keep these images to check later the correctness of the parallel versions of the other sections. Now we will use Tareador to see the graph dependencies using both solvers:

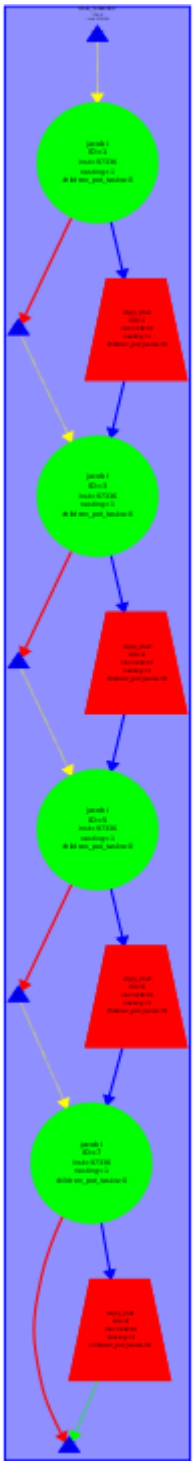


Figure 3: Jacobi TDG

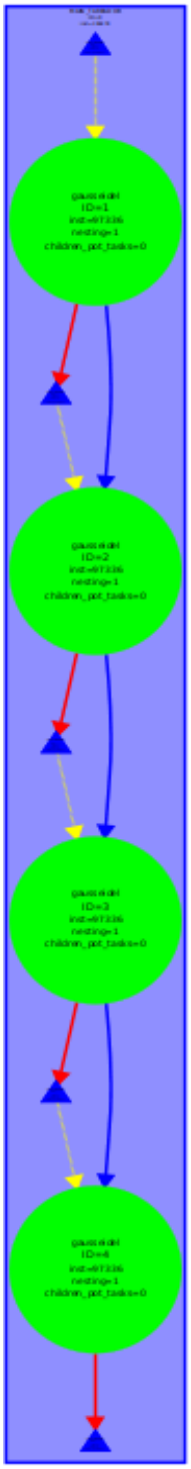


Figure 4: Gauss-Seidel TDG

Is there any parallelism that can be exploited at this granularity level?

In the Jacobi version, we can see that the function copy_mat creates big dependencies in the TDG, because it's how this version works, copying the content of param.u to a temporary matrix (param.help). The dependencies between the Jacobi (green circle) and copy_mat (red trapezium) functions come back and forth. The parallelism that could be exploited would be from the Jacobi to the copy_mat and MAIN_TAREADOR functions, each one using a thread. However, this would be very imbalanced, so this wouldn't be the best option.

In the Gauss-Seidel version, we get rid of the copy_mat function as we only work with a single matrix (param.u). The big green circles (Gauss-Seidel functions) are dependent on the previous Gauss-Seidel function, because the program calculates the final result in blocks. For this version, we don't think parallelism would improve the program.

To explore a finer granularity for both solvers we changed the solve function adding tareador_start_task() and tareador_end_task() after the second loop to create one task per block, and with the Dataview option in Tareador we have identified that the variable "sum" was the one causing serialisation, so then we added tareador_disable_object() and tareador_enable_object() to protect the access in the implementation and to "ignore" it so it can be parallelised:

Here are the changes we made in the solve-tareador code:

```
// 2D-blocked solver: one iteration step
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) {
        tareador_start_task("task");
        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;
          }
        }
        tareador_end_task("task");
      }
    }
    tareador_enable_object(&sum);

    return sum;
}
```

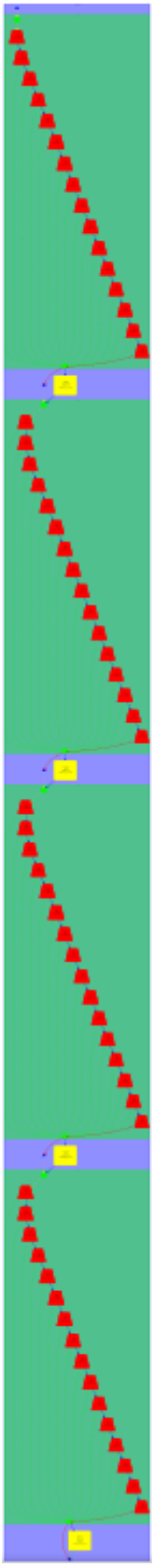
Figure 5: solve-tareador.c changed

```
(param->u)     = (double*)calloc( sizeof(double),np*np );
(param->uhelp) = (double*)calloc( sizeof(double),np*np );
```

Figure 6: misc.c lines 37 and 38

The copy of data between the two matrices is what causes the dependencies. The Gauss-Seidel solver uses mostly line 37 and the Jacobi uses both lines over and over again when going from a green task to a red one and vice versa.

Jacobi solver without protecting “sum” variable:



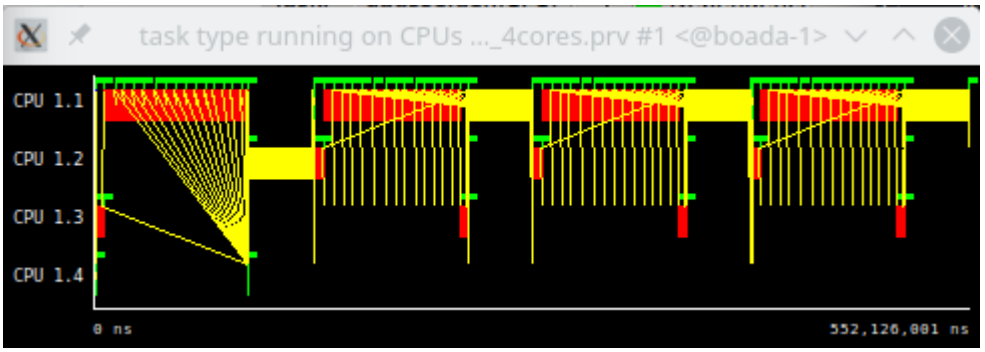


Figure 8: Thread state of Jacobi solver in Paraver

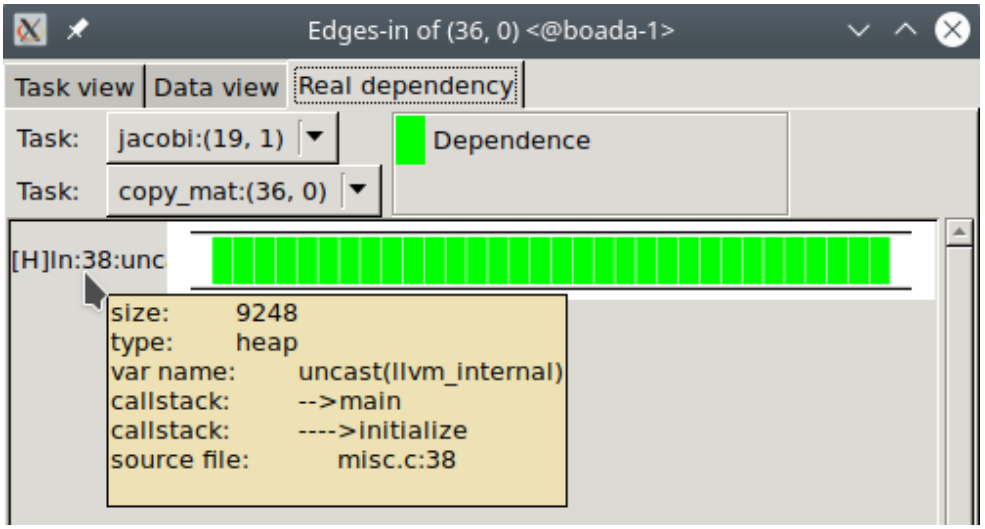


Figure 9: Dataview option in Tareador

We can see that it is serialised and there is no parallelism at all (figure 7) and there is a lot of time wasted in the yellow parts of the figure 8, which are the copy_mat parts (not parallelised for now), so it is really improvable.
The explanation of figure 9 is done in figure 6.

Figure 7: TDG

Gauss-Seidel solver without protecting “sum” variable:

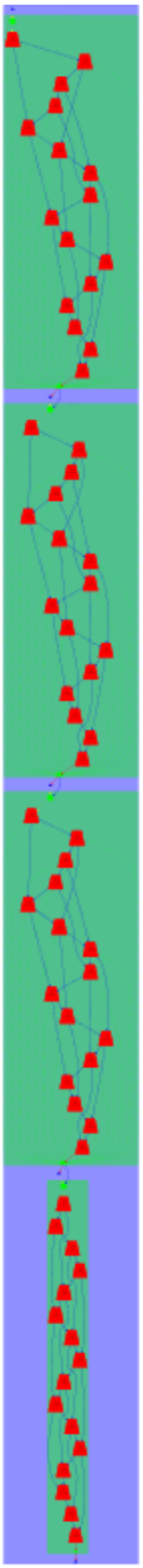


Figure 10: TDG

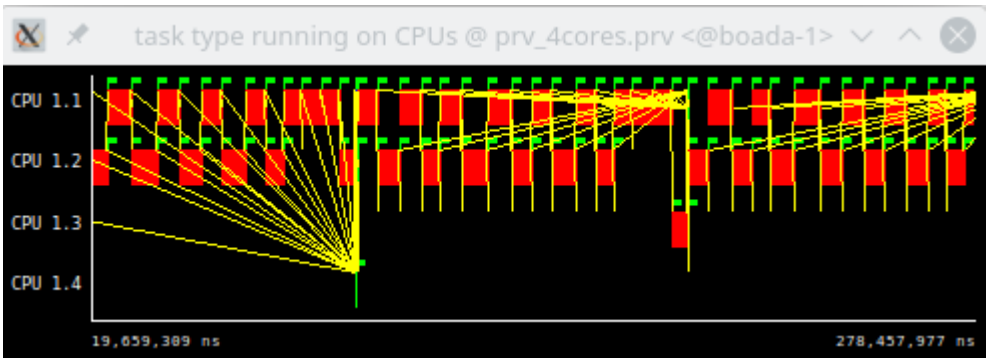


Figure 11: Thread state of parallelised Gauss-Seidel solver in Paraver

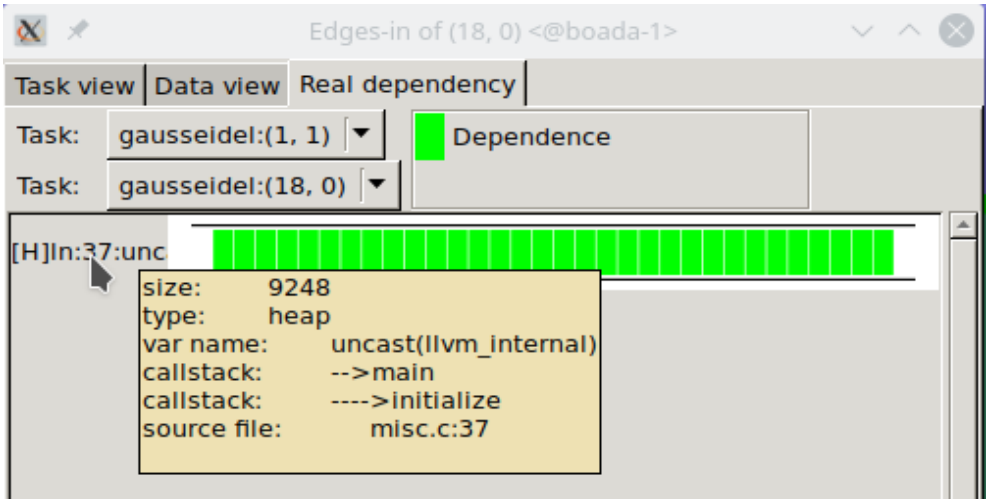


Figure 12: Dataview option in Tareador

This version is much better than the previous Jacobi version since there is no copy_mat function and there are only red parts, which are the tasks in question.
The explanation of figure 12 is done in figure 6.

Jacobi solver protecting “sum” variable:

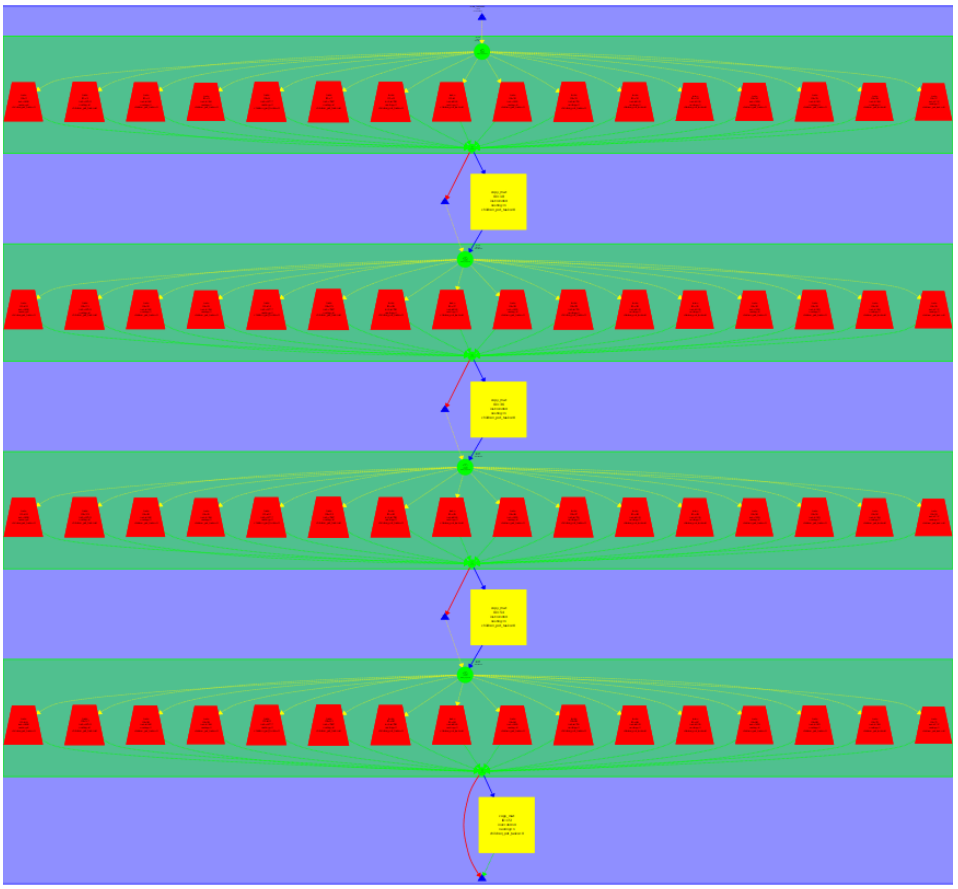


Figure 13: TDG

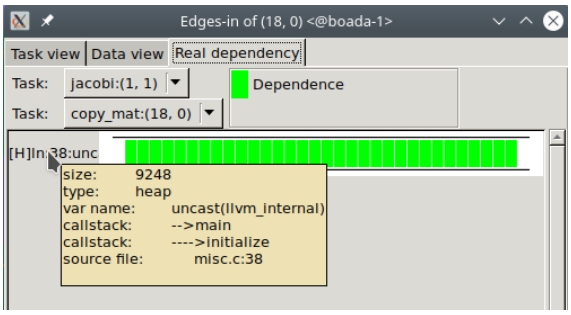


Figure 14: Dataview option in Tareador

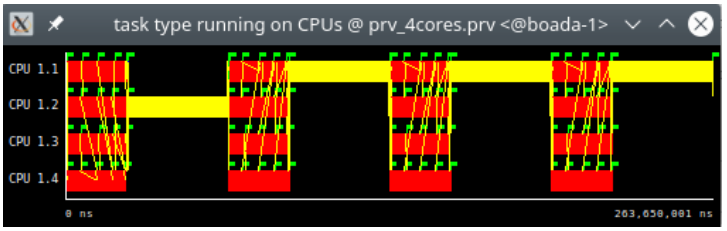


Figure 15: Thread state of Jacobi solver in Paraver

After uncommenting the tareador_disable_object(&sum) and tareador_enable_object(&sum) parts, we can claim that the figure 13 is much better parallelised than the figure 7, although it can even be better since the yellow copy_mat parts are still a problem and have not been parallelised yet. The explanation of figure 14 it is done in figure 6.

Gauss-Seidel solver protecting “sum” variable:

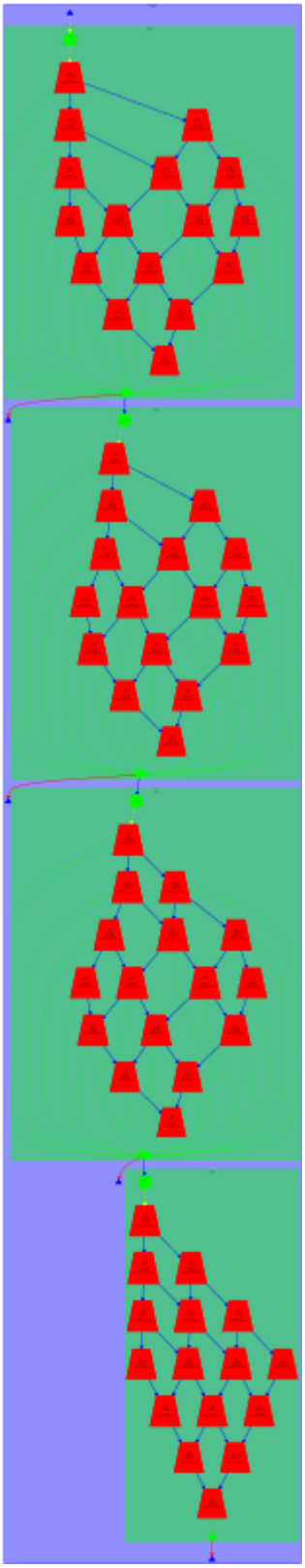


Figure 16: TDG

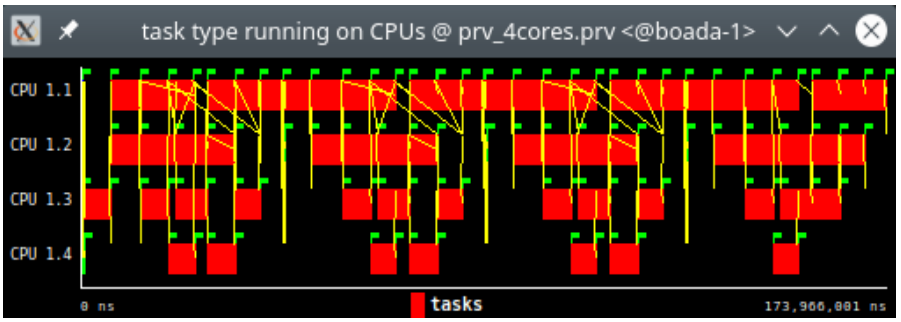


Figure 17: Thread state of parallelised Gauss-Seidel solver in Paraver

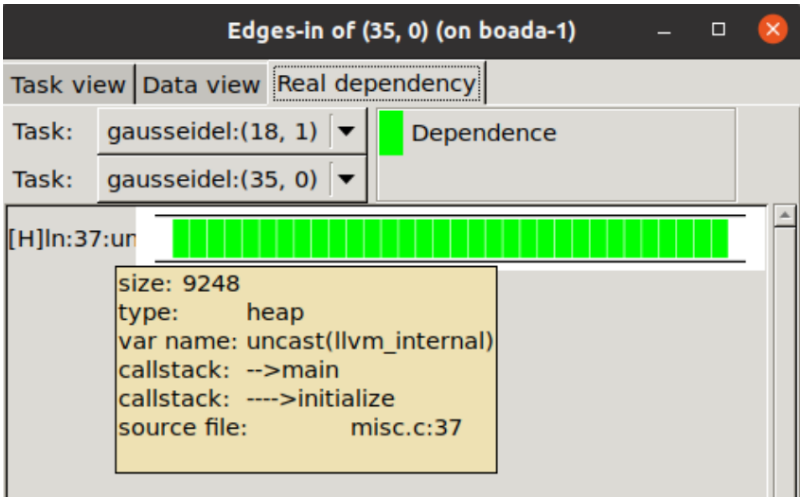


Figure 18: Dataview option in Tareador

After protecting the variable “sum” there is also improvement in the parallelism of the Gauss-Seidel solver but we do not think there is much more left to be parallelised.
The explanation of figure 18 is done in figure 6.

Is there any other part of the code that can also be parallelised?. If so, modify again the instrumentation to parallelise it.

Having seen the Tareador and Paraver dependencies we can conclude that the Gauss-Seidel solver can not be parallelised much more since there is no yellow part (figure 17) to be parallelised like the other one, but like we have commented before, there is a lot left in the Jacobi solver (figure 15), so we changed the other function the latter solver uses and we executed it in Tareador and Paraver and now we can finally say that it is parallelised:

```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=4;
    int nblocksj=4;

    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) {
        tareador_start_task("task");
        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];
        tareador_end_task("task");
      }
    }
}
```

Figure 19: copy_mat in solve-tareador.c changed

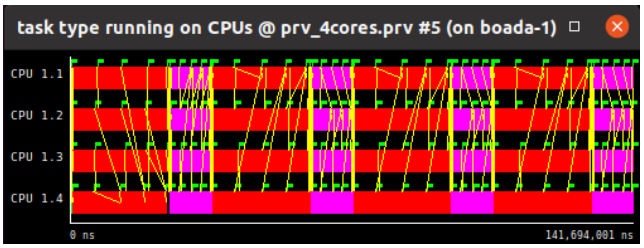


Figure 20: Thread state of parallelised Jacobi solver in Paraver

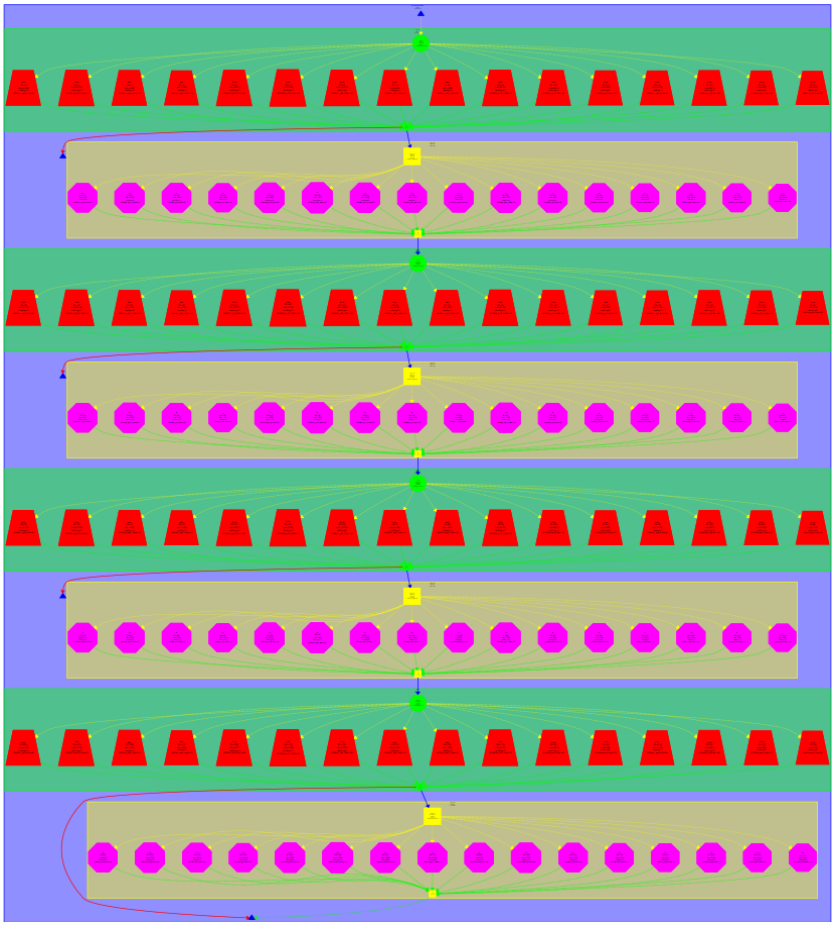


Figure 21: TDG

3. Parallelisation of the heat equation solvers: Jacobi solver

Considering the Jacobi solver, we used the implicit tasks generated in #pragma omp parallel, following a geometric block data decomposition by rows to parallelise the code.
We completed it by adding reduction(+:sum) private (tmp, diff), since the variable "sum", as we have seen in the previous session, needs to be protected and as it equals to diff*diff and "diff" needs the variable "tmp", we have put them into private clauses:

```
// 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=1;

    #pragma omp parallel reduction(+: sum) private(tmp,diff) // complete data sharing constructs here
    {
      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 21: solver-omp.c changed (Jacobi version)

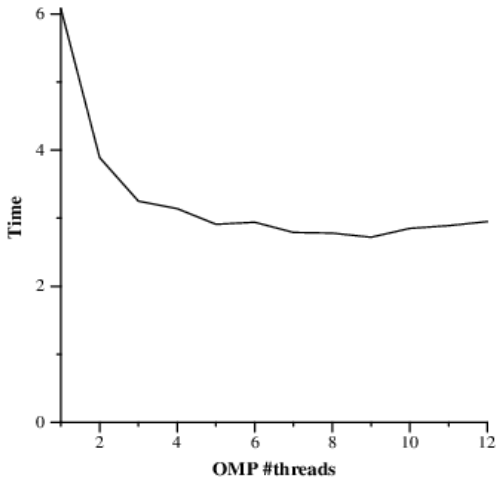
By compiling and executing it with sbatch and 8 threads, we have obtained the following results:

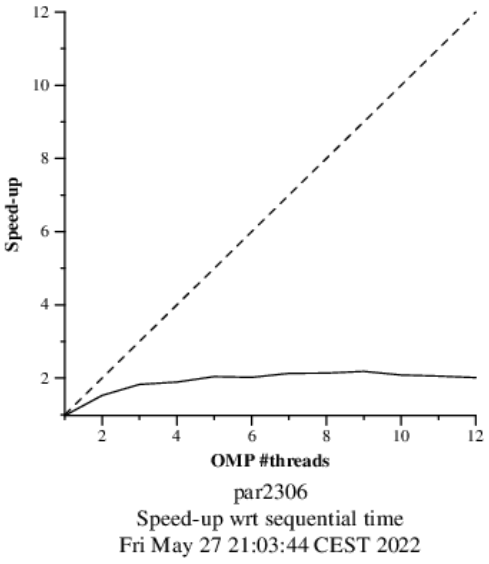
Iterations : 25000
Resolution : 254
Residual : 0.000050
Solver : 0 (Jacobi)
Num. Heat sources : 2
1: (0.00, 0.00) 1.00 2.50
2: (0.50, 1.00) 1.00 2.50
Time: **3.088**
Flops and Flops per second: (11.182 GFlop => 3620.99 MFlop/s)
Convergence to residual=0.000050: 15756 iterations

Figure 22: display heat-jacobi.ppm

Comparing this to the figure 1 we can notice that the execution time has decreased, and it seems like that it is the same image.

Once validated, we checked the scalability plots from 1 - 12 processors and as we suspected, we still need to parallelise the function copy_mat since the plots are far to be nice:





Figures 23 and 24: Jacobi solver speed-up plots

Then we also analysed the execution of the binary with Paraver and we can see that every thread runs the execution (blue parts) with little black spaces that tells us that they are waiting for the main thread (thread 1) to finish the execution of the sequential parts in order to continue the execution of the parallelised ones. The main thread also takes responsibility of the creation and the synchronization of the tasks:

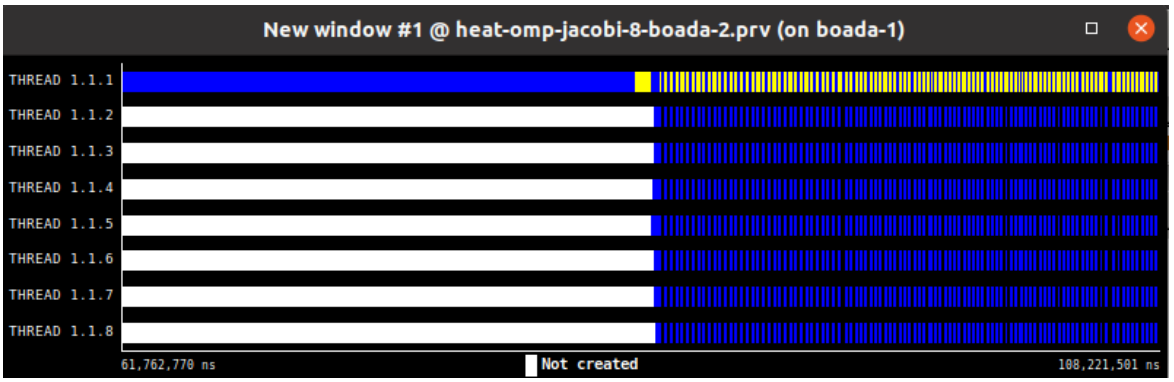


Figure 25: Thread state of Jacobi solver half parallelised

Now we parallelised the other function copy_mat as well, adding some modifications to the code and the clause #pragma omp parallel:

```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    #pragma omp parallel
    {
      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(blocki, 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 26: copy_mat in solver-omp.c changed (Jacobi version)

After executing it we can see that it gave us better results, as we suspected before:

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: **0.736**
Flops and Flops per second: (11.182 GFlop => 15196.49 MFlop/s)
Convergence to residual=0.000050: 15756 iterations

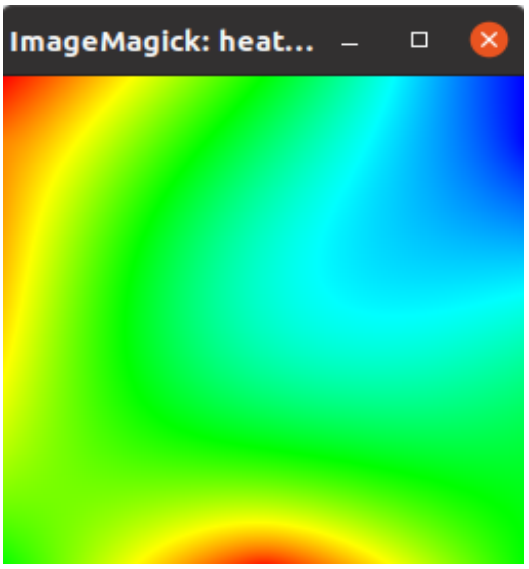


Figure 27: display heat-jacobi.ppm

The image seems to maintain its colours and we can see that it does not differ from the other figures (1 and 22) using diff command but there is a change that we need to appreciate, and that is the execution time. It has improved a lot (from 3 seconds to 0,7 seconds approximately) and it is more efficient now since copying a matrix can be really expensive, and parallelising its code and having these results demonstrates that it is true.

Once more, we validated the Paraver outcome and now we can see that the little black spaces (waiting time) have almost all gone:

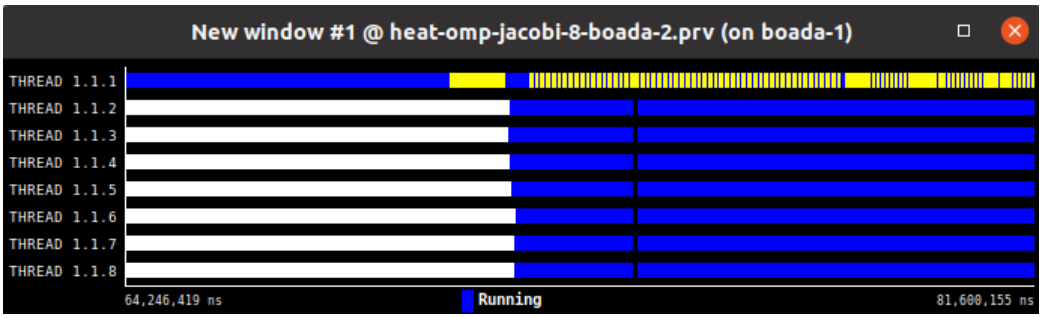
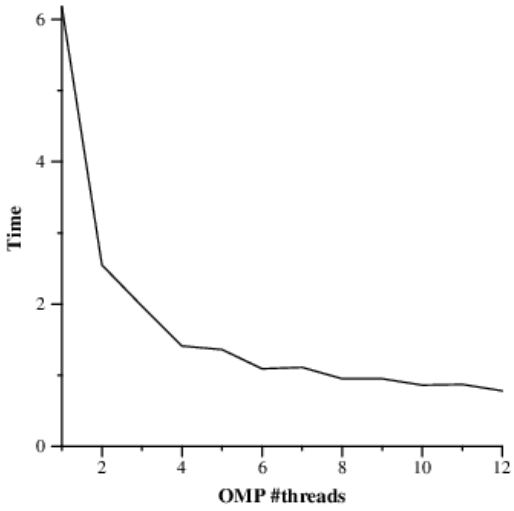
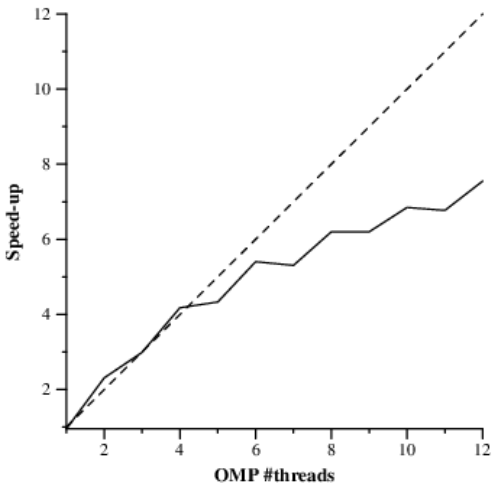


Figure 28: Thread state of Jacobi solver fully parallelised

And finally we can see that the scalability plots have enormously improved: the execution time has reduced a lot and the speed-up has increased by far, now exploiting the parallelism much better.





Figures 29 and 30: Jacobi solver speed-up plots

3. Parallelisation of the heat equation solvers: Gauss-Selder solver

Now considering the Gauss-Seidel solver, the big change has been in the solve function, we have added a simple if at the beginning of the program, saying that if u == unew, if it returns true, this will mean that we are using this solver, otherwise we would be using the Jacobi, with 2 different matrices (figure 32).

Then in the body of the function, we have created a parallel region, with the variables tmp and diff privatised, like the previous case, the variable next shared, so all threads share the same matrix. This method does a geometric block data decomposition by columns, and this matrix helps us parallelise the code. Finally, we keep the reduction of the variable sum, like in the Jacobi solver.

```
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*userparam;
        next[0][0] = nblocksj;
        for (int i = 1; i < nblocksi; i++) next[i][0] = 0;

        #pragma omp parallel private(tmp, diff) shared(next) firstprivate(nblocksi, nblocksj)
reduction(+: sum)// complete data sharing constructs here
        {
        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); // waiting to advance to the next block

            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;
                  #pragma omp atomic write
                    unew[i*sizey+j] = tmp;
                }
            }

            if (blocki < nblocksi-1) {
                #pragma omp atomic update
                next[blocki+1][0]++;
            }
        }
        }
    }
```

Figure 31: First part of solver-omp.c (Gauss Version)

```
else{
    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    #pragma omp parallel private(tmp, diff) firstprivate(nblocksi, nblocksj) reduction(+:
    sum)// complete data sharing constructs here
    {
    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;
```

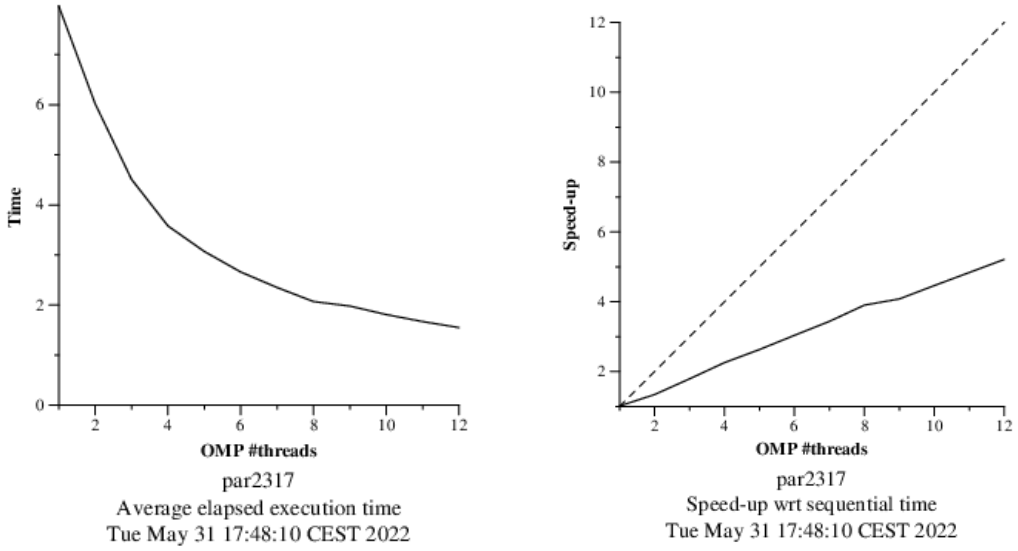
Figure 32: Second part of solver-omp.c (Gauss Version)

After compiling the program, executing it and checking that it returns the same output as in the sequential version (diff), we get the following results:

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: **1.680**
Flops and Flops per second: (8.806 GFlop => 5242.01 MFlop/s)
Convergence to residual=0.000050: 12409 iterations

Figure 33: display heat-gausseidel.ppm

After this, we have plotted the scalability graphs, and the result looks like this:



Figures 34 and 35: Scalability plots of the Gauss Version

As we can see, the overall performance of the program needs to be improved. It deviates quickly from the ideal performance. But this happens because we can not parallelise the copy_mat function unlike the previous solver.

If we generate traces with Paraver, we can see that the main thread is the one in charge of creating the tasks, while the others do the work. A key difference with the Jacobi solver is that we don't see black spaces in the threads when they synchronize (right below the yellow parts), because this method doesn't use an auxiliary matrix, we read and write in the same one.

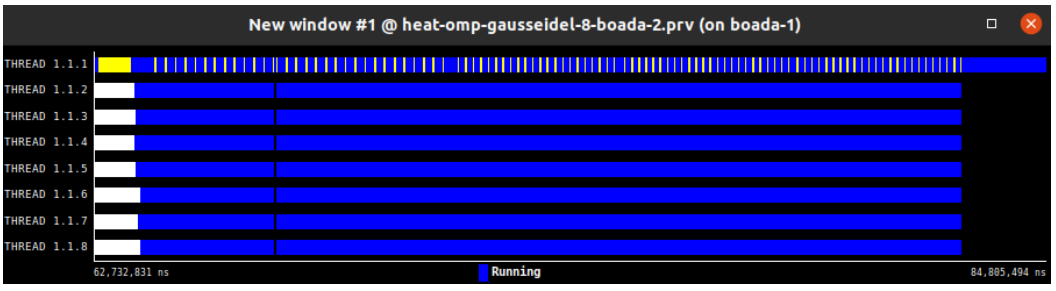
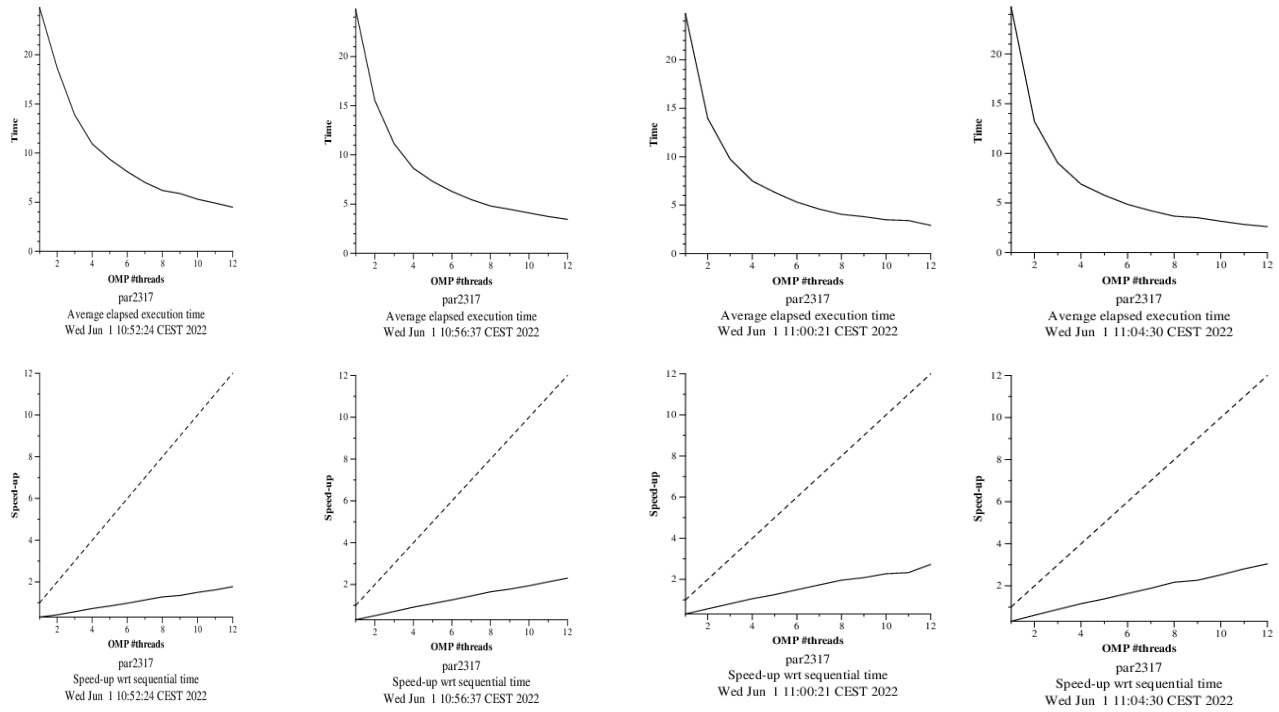


Figure 36: Thread State of solver-omp.c (Gauss Version)

Now we will change the code a bit more to further exploit the parallelism. We will change the number of blocks in the j dimension by modifying the declaration of the nblocksj variable. We will declare it with the same value as userparam multiplied by nblocksi. As we can see in the following outputs, changing its value reduces the execution time of the program.

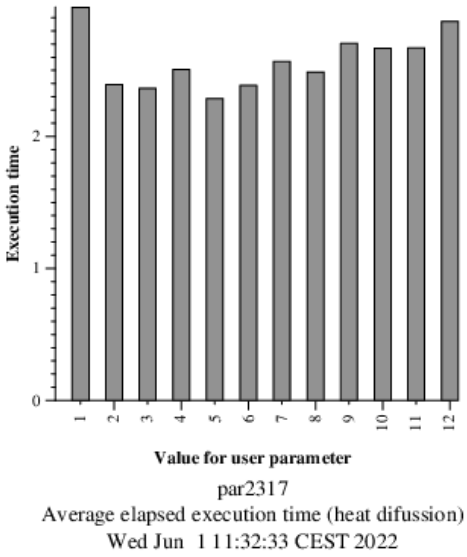
The following scalability plots show us that changing the value of userparam reduces its execution time and increases the speed-up:

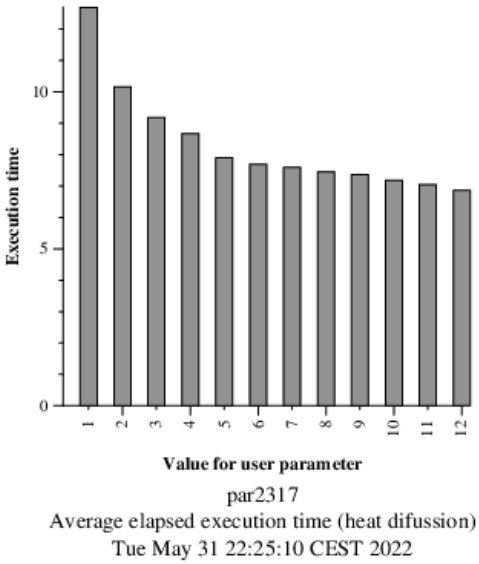


Figures 37,38,39 and 40 (vertical): Scalability plots with userparam=1,2,4 and 8

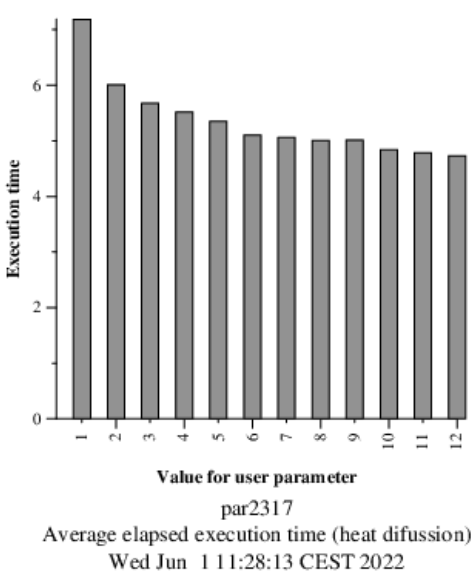
It is not a big win in speedup, but we can see that the execution time and speedup of userparam = 12 is much better than with userparam = 2.

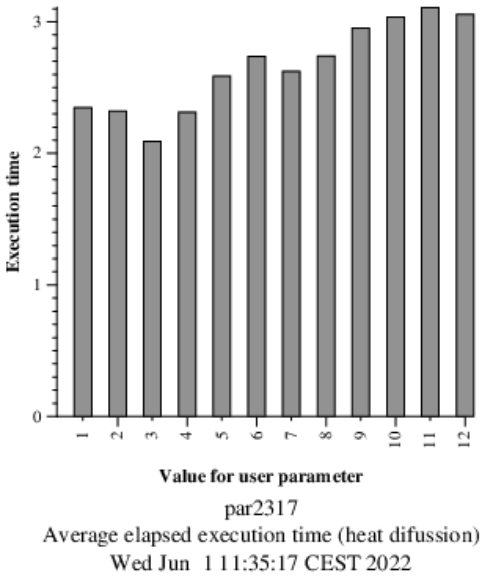
Finally, in order to get the best value of userparam, we have executed the submit-userparam-omp.sh script with 4 and 8 threads. The plots look like this:





Figures 41and 42: User param plots with 8 and 4 threads





Figures 43 and 44: User param plots with 2 and 12 threads

As we can see here, the execution time of the program changes depending on the value of userparam. In conclusion, high values of userparam work better with low number of threads and the other way around, for example with 12 threads, the ideal value is small, as the execution time starts to rise once this value rises too.

5. Optional 1

Now we are going to use explicit tasks instead of implicit ones for the Jacobi solver, and follow an iterative task decomposition. It is been added #pragma omp parallel followed by #pragma omp single in both functions and the clause #pragma omp taskloop before the first loop in the solve function:

```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    #pragma omp parallel
    #pragma omp single // optional 1
    {
      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];
      }
    }
}

// 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=1;

    //#pragma omp parallel reduction(+: sum) private(tmp,diff) // complete data sharing constructs here
    #pragma omp parallel // optional 1
    #pragma omp single // optional 1
    {
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      #pragma omp taskloop
      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;
          }
        }
```

Figure 45 : solver-omp.c for optional 1

Once compiled and executed we obtain that the execution time is really nice as it is 0,221 seconds.

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: **0.221**
Flops and Flops per second: (2.754 GFlop => 12440.34 MFlop/s)
Convergence to residual=0.000049: 3881 iterations

Then in Paraver results we can see that all threads are almost just scheduling and doing synchronization:

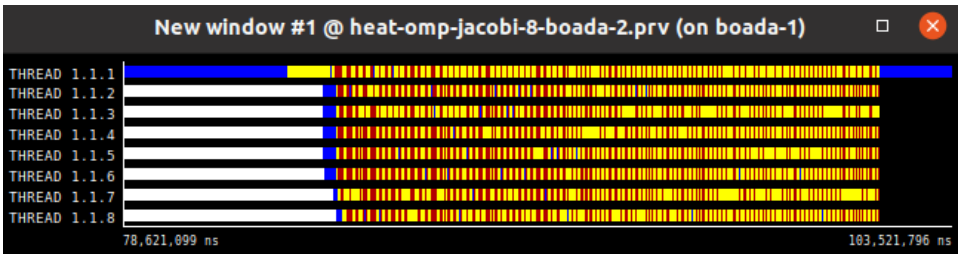
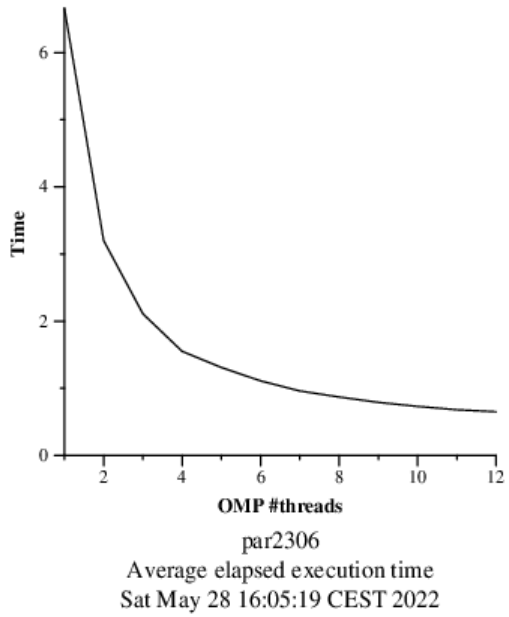
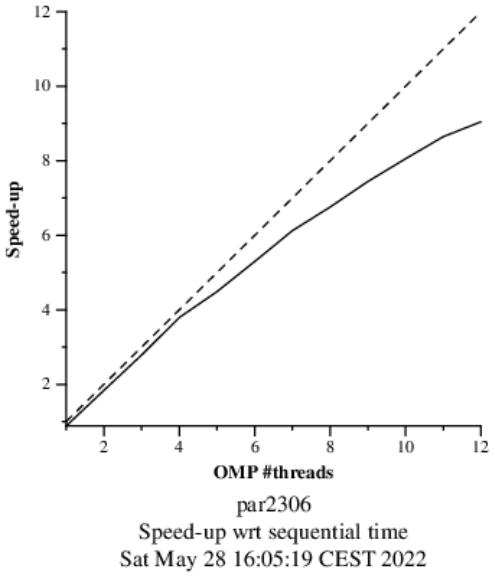


Figure 46:Thread state of Jacobi solver with explicit tasks

With the scalability plots we can see that the execution time reduces considerably with the increase of the threads and the speed-up it is nearly ideal:





Figures 47 and 48: Jacobi solver speed-up plots

6. Conclusion

The first day, we became familiar with both solvers, as we got to understand how they worked, what their differences were, and we compared them with their TDGs. We found out that the Jacobi version copies the data it's working with to an auxiliary matrix, while the Gauss-Seidel version works in the same matrix, overwriting data. We found the dependencies of the programs with the help of the TDGs and the Paraver traces.

The second day, we managed to parallelise both solvers using OpenMP clauses following geometric block data decompositions. In the Jacobi solver, the most efficient way was doing it by rows, so each block would be a set of adjacent rows. Later we measured its scalability and we kept parallelising it by adding more clauses to its functions. On the other hand, with the Gauss-Seidel version, we followed a different strategy, by columns. We repeated the process like the previous version, and we used the userparam variable to check how the execution time of the program changed depending on its value.