Parallelism

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

Marian Danci & David Valero
Group 42, par4207
Fall 2018-19

Index

Introduction

In this lab we will work with a program which simulates heat diffusion in a solid body using two different algorithms for the heat equation, Jacobi and Gauss-Seidel, each one has different numerical properties and shows different parallel behaviours. The purpose of the algorithm is to do the processing of each pixel, from the average of the adjoining cells.

In the following picture, we can see the file 'heat.ppm' providing the solution as an image (as a portable pixmap file format) that the program generates.The program is executed with a configuration file 'test.dat' that specifies the maximum number of simulation steps 'iterations', the size of the body 'resolution', the solver to be used 'algorithm' and the heat sources, their position, size and temperature. In this case, we have two heat sources are placed in the borders of the 2D solid, one in the upper left corner and the other in the middle of the lower border.

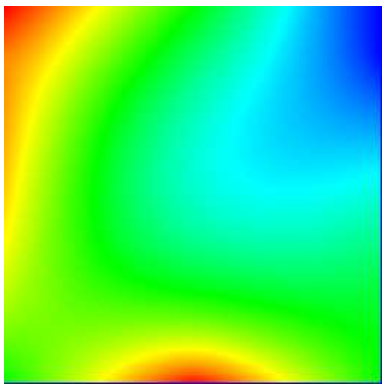


Image 1: Image representing the temperature in each point of the 2D solid body.

Session 1

1.1: Sequential heat diffusion program

After compiled using `make heat` and executed the program with `./heat test.dat`, we obtained the following output:

```
par4207@boada-1:~/lab5$ ./heat test.dat
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 5.813
Flops and Flops per second: (11.182 GFlop => 1923.67 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

In the output we can see the following report:

- `Time`: the execution time in seconds.
- `Flops and Flops per second`: the number of floating point operations (Flop) performed and the average number of floating point operations performed per second (Flop/s).
- `Convergence to residual`: the residual and the number of simulation steps performed to reach that residual.

In addition to the following image:

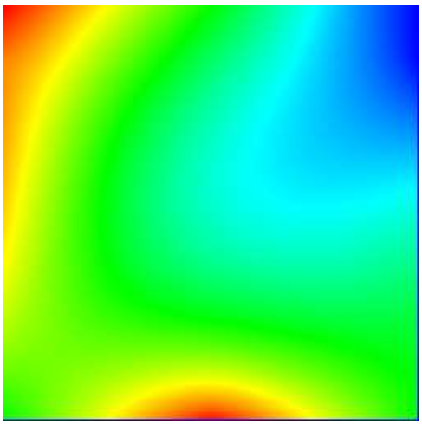


Image2: heat.ppm with Jacobi

And to obtain the result with the Gauss-Seidel algorithm we changed in `test.dat` the '0' value, which specifies to use Jacobi, into '1':

```
1         # Algorithm 0=Jacobi 1=Gauss-Seidel
```

Obtaining the following output:

```
par4207@boada-1:~/lab5$ ./heat test.dat
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 6.324
Flops and Flops per second: (8.806 GFlop => 1392.54 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

And this image:

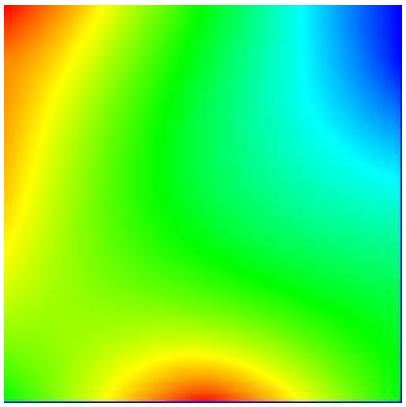


Image3: heat.ppm with Gauss-Seidel

Finally, we checked that the images generated when using the two solvers are different, with the 'diff' command:

```
par4207@boada-1:~/lab5/S1$ diff img-heat-jacobi.ppm
img-heat-gauss.ppm
```

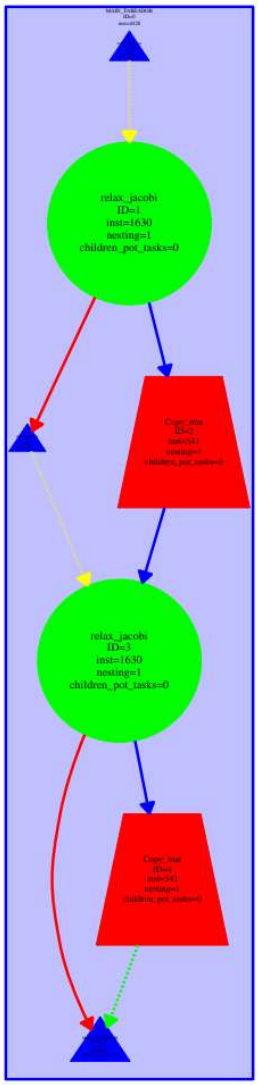
Output:

```
Binary files img-heat-jacobi_ppm.png and
img-heat-gauss_ppm.png differ
```

1.1: Analysis with Tareador

In this section, we will use Tareador to analyze the task graphs generated when using the two different solvers. To configure it we use `small.dat` file instead of `test.dat`, which just performs a couple of iterations on a very small image.

After compiling the `heat-tareador` file with `make heat-tareador` and executing it with `run-tareador.sh` script, we obtained the following depend graph:



Graph1: Original depend graph for Jacobi.

As we can see, the tasks are huge, each matrix is a task, so is difficult to obtain an analysis with so few tasks. To improve it, we explore the dependencies that happen when a much finer-grain task decomposition is used, putting one task for each iteration of the body of the innermost loop. To do it, we change the original Tareador instrumentation to reflect the newly proposed task granularity, putting a `tareador_start_task("task_name")` and `tareador_end_task("task_name")` when `copy_Mat(...)` is called, as we can see in the code below:

```
tareador_ON();
// starting time
runtime = wtime();

iter = 0;
while(1) {
switch( param.algorithm ) {
        case 0: // JACOBI

            //tareador_start_task("relax_jacobi");
            residual = relax_jacobi(param.u, param.uhelp, np, np);
            //tareador_end_task("relax_jacobi");

            // Copy uhelp into u
            tareador_start_task("Copy_mat");
            copy_mat(param.uhelp, param.u, np, np);
            tareador_end_task("Copy_mat");

            break;
        case 1: // GAUSS

            //tareador_start_task("relax_gauss");
            residual = relax_gauss(param.u, np, np);
            //tareador_end_task("relax_gauss");

            break;
    }

    iter++;

    // solution good enough ?
    if (residual < 0.00005) break;

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

// Flop count after iter iterations
flop = iter * 11.0 * param.resolution * param.resolution;
// stopping time
runtime = wtime() - runtime;
tareador_OFF();
```

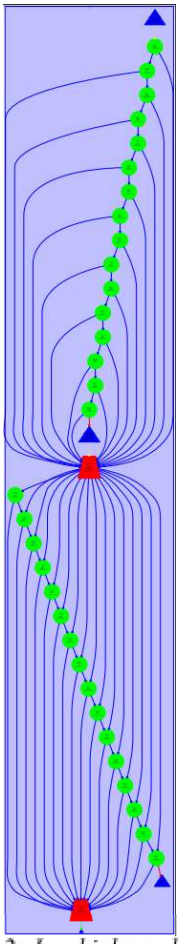
Code1: heat-tareador.c Jacobi with changes.

Furthermore, in the `relax_jacobi(...)` function, we put in the code of the inner loop the `tareador_start_task("task_name")` and `tareador_end_task("task_name")` clauses, as you can see in the following code:

```
/*
 * Blocked Jacobi solver: one iteration step
 */
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("relax_jacobi");
         utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
                                   u[ i*sizey     + (j+1) ]+  // right
                       u[ (i-1)*sizey + j     ]+  // top
                       u[ (i+1)*sizey + j     ]); // bottom
         diff = utmp[i*sizey+j] - u[i*sizey + j];
         sum += diff * diff;
         tareador_end_task("relax_jacobi");
     }
      }
    }

    return sum;
}
```

Code2: solver-tareador.c with the changes

We compile again and after executing it we obtained the *Graph2*.



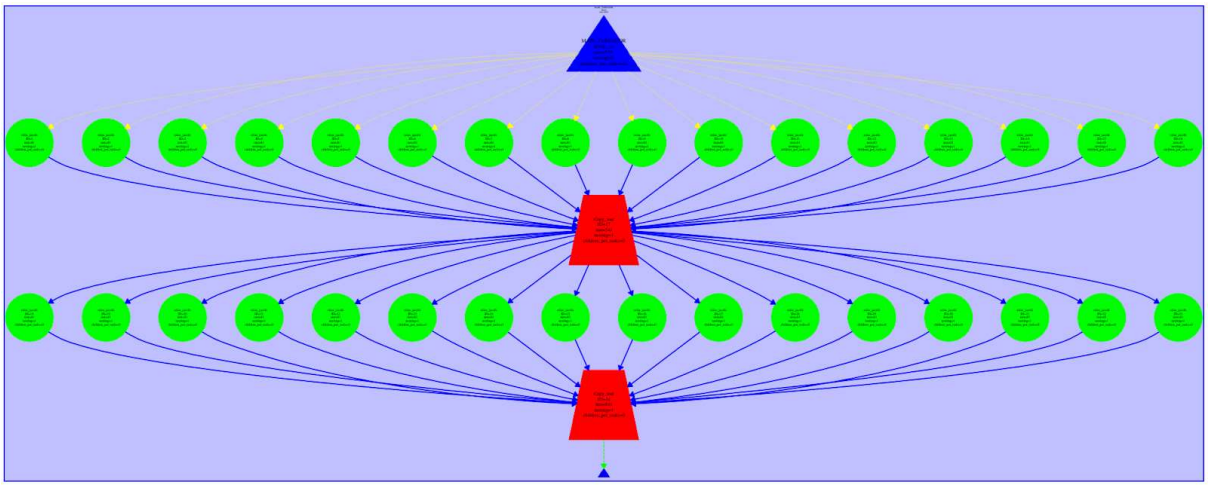
Graph2: Jacobi depend graph

As we can see, there are some variables that are causing a great serialization of all the tasks, making its parallelization impossible.
To find it out, we did a right click, putting the date-view, and we detect there is an intersection in variable `sum`, concluding that is variable `sum` which is causing this serialization.

To solve it, we put the following instruction in the code to remove the dependency:

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

After that, we compile and execute it again and we obtain *Graph3*.



Graph3: Jacobi depend graph - without sum dependence.

As we can see, the parallelization has increased a lot because almost all the tasks can be done at the same time. Only the Copy of the matrix tasks has dependencies and need to be done alone.

This dependence can be solved it in OpenMP with the `#pragma omp for reduction(+: sum)` clause, which makes that each thread has a private variable and when the iterations finish the `sum` value of each thread it accumulates in the original variable `sum`.

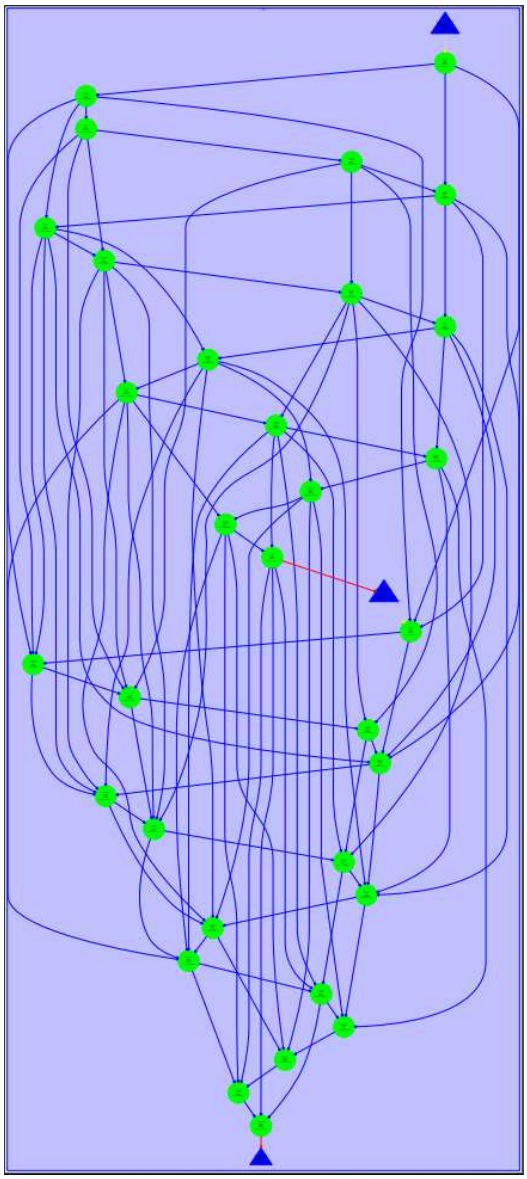
We repeat the process with the Gauss-Seidel solver, changing the `small.dat` file to Gauss algorithm and the Gauss block in `solver-tareador.c`.

As we did before, we change the `tareador_start_task("task_name")` to increase the number of tasks:

```
/*
 * Blocked Gauss-Seidel solver: one iteration step
 */
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("relax_gauss");
        unew= 0.25 * ( u[ i*sizey   + (j-1) ]+  // left
               u[ i*sizey   + (j+1) ]+  // right
               u[ (i-1)*sizey   + j     ]+  // top
               u[ (i+1)*sizey   + j     ]); // bottom
        diff = unew - u[i*sizey+ j];
        sum += diff * diff;
        u[i*sizey+j]=unew;
        tareador_end_task("relax_gauss");
        }
      }
    }

    return sum;
}
```

Code3: solver-tareador.c, Gauss with changes.

Obtaining the following dependence graph:



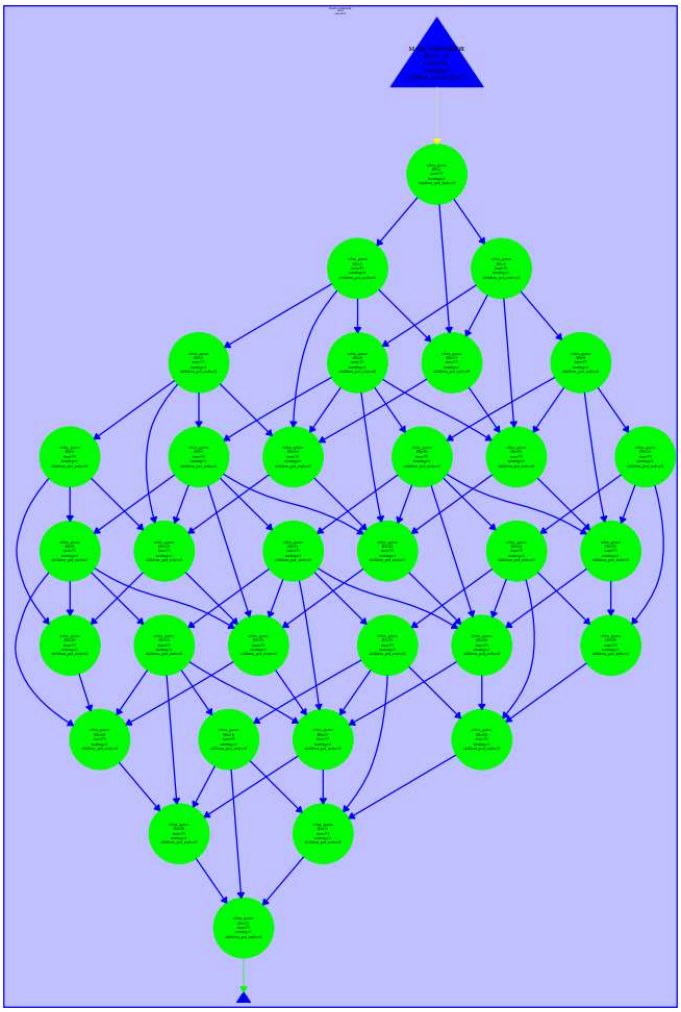
Graph4: Gauss-Seidel depend graph

We can see in the graph, that all tasks are made sequentially because of the dependences.
As we did in the Jacobi version, we can solve the dependence in `sum` with `#pragma omp for reduction(+ : sum)` clause. Another solution could be with the `#pragma omp task` with a `depende()` clause in which we would specify the dependencies in the matrix.

Moreover, as in Jacobi, to solve the dependences we put the following instruction:

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

After all these changes, we obtain the following graph:



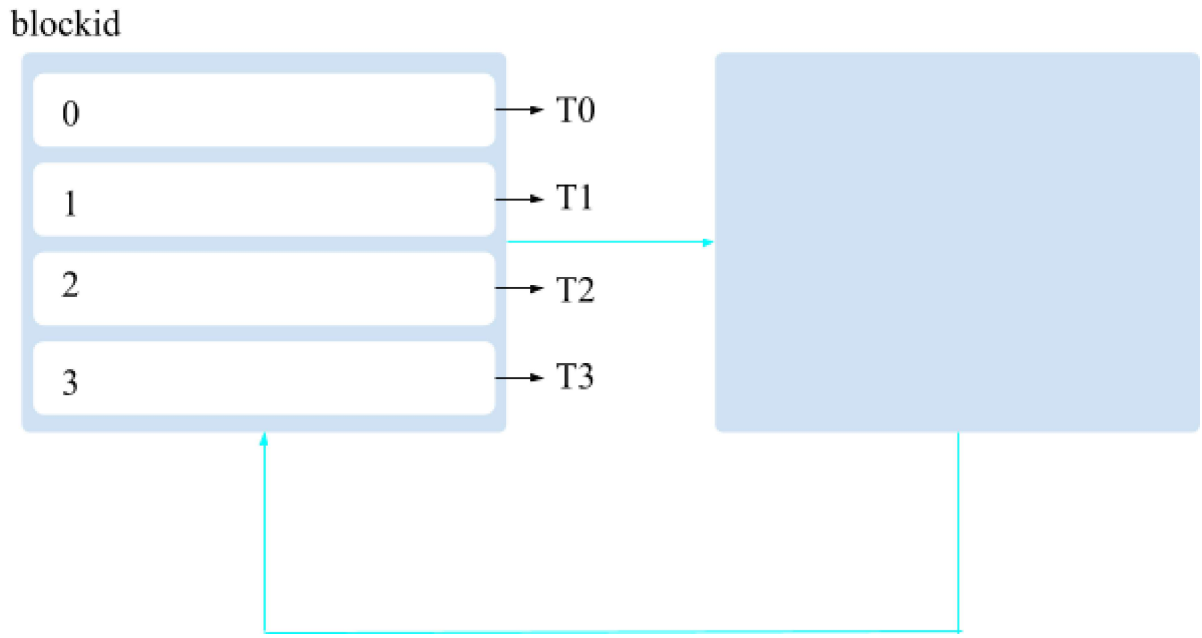
Graph5: Gauss depend graph - without sum dependence.

In conclusion, in Jacobi there is a dependence caused by `sum` variable, but in Gauss there are also dependencies in the matrix since there is not an auxiliary matrix as in Jacobi. In the depend graph above we can see that begins with a bit parallelization, in the middle reaches its maximum and the second half is symmetrical.

Session 2: Parallelization of Jacobi with OpenMP `parallel`

In this section, you will parallelize the sequential code for Jacobi using `#pragma omp parallel`, following a given geometric data decomposition.
To start with, we make a copy of the sequential heat.c and solver.c into heat-omp.c and solver-omp.c, respectively.

Then, we draw the geometric data decomposition that is generated when these macros are used in the Jacobi solver for howmany=4, obtaining *Picture1*.



Picture1: Geometric data decomposition in Jacobi solver.

As is seen, we divided the matrix into different packs of rows (blocks), each one with a different blockid. These values of each group are saved in a different auxiliary matrix, that allows to recover the values when necessary to the original matrix. And each block is assigned to a thread.

Next, we parallelize the code in function `relax_jacobi(...)` following this geometric data decomposition.
First of all, we put a `#pragma omp parallel for private(diff) reduction(+:sum)` clause at the start of the function, allowing the parallelisation, making `diff` private and making a reduction when we add values in `sum` variable,
Then, we analyze `lowerb` and `upperb` values (in heat.h), that delimit the value of "i" for each loop, where each group is one different blockid.

Resulting in the following code:

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

    int howmany=4;

    #pragma omp parallel for private(diff) reduction(+:sum)
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
         utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
                                   u[ i*sizey     + (j+1) ]+  // right
                      u[ (i-1)*sizey + j     ]+  // top
                      u[ (i+1)*sizey + j     ]); // bottom
         diff = utmp[i*sizey+j] - u[i*sizey + j];
         sum += diff * diff;
     }
      }
    }

    return sum;
}
```

Code4: Jacobi block solver-omp.c

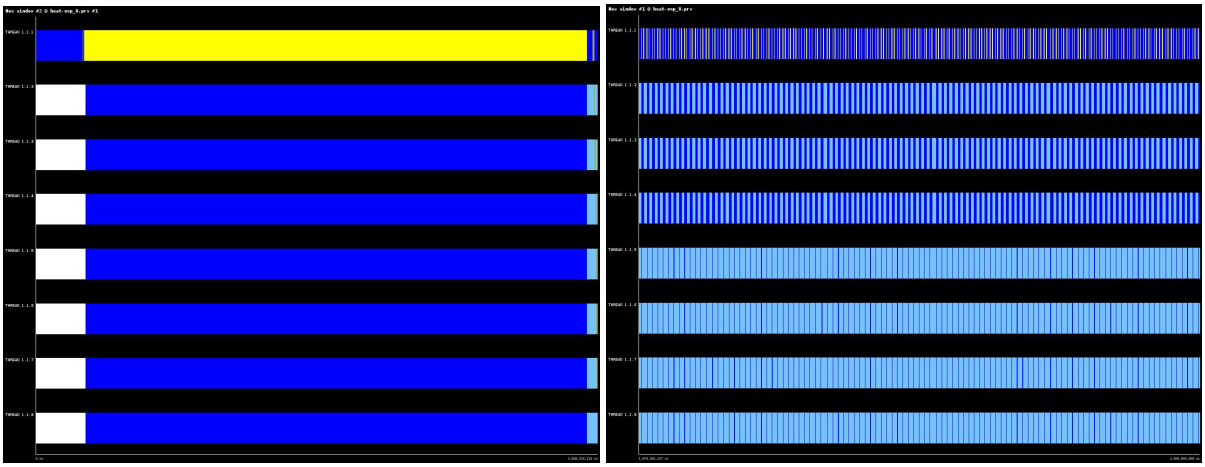
We compile using `make heat-omp` and submit its execution to the queue using the submit-omp.sh script (using 8 threads):

```
par4207@boada-1:~/lab5$ qsub -l execution submit-omp.sh
```

After that, we could validate the parallelization by visually inspecting the image generated and making a diff with the file generated with the original sequential version. No difference was found.
We obtain the following output with `howmany=4` in heat-omp_8.times.txt:

```
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 3.169
Flops and Flops per second: (11.182 GFlop => 3528.91 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

After that, we instrument the execution of the binary with *Extrae* by submitting the `submit-omp-i.sh` script to boada. And we obtain the following traces:



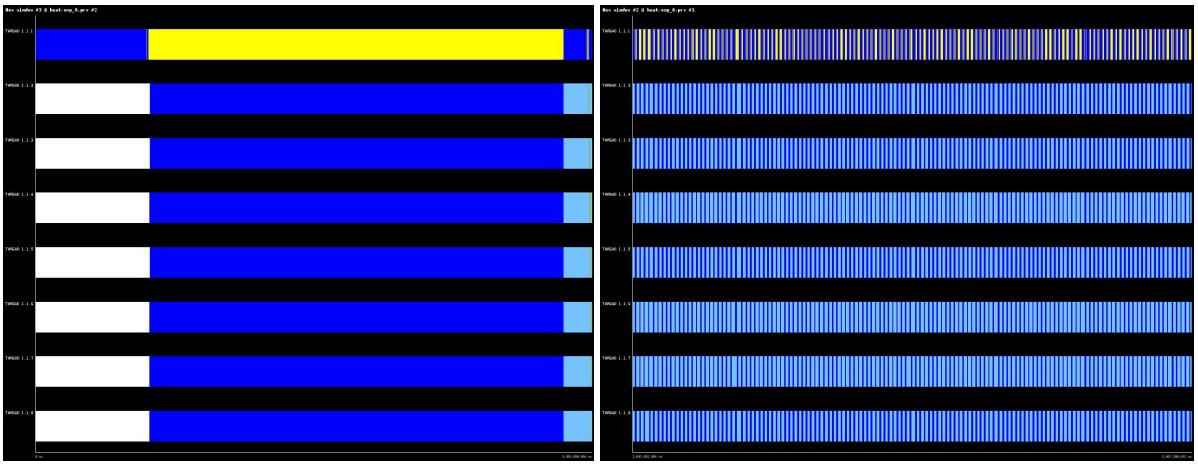
Picture2: Traces in Jacobi solver. Left: original. Right: zoom.

In the first picture we can see the complete trace and in the second one, we zoomed 4 times to the centre of the trace to appreciate the behaviour.
As is seen in the zoom-it trace, only the first four threats are doing tasks, the other fours are working a lot less.
So, we can conclude that the distribution of work is not adequate for 8 threads because in this case with a `howmany=4` only 4 threads do the work in parallel and the others do not take advantage of the parallelism. We have a load balancing problem.
To avoid this bottleneck we assign to the variable `howmany` the number of threads (with the clause `omp_get_max_threads()`), so each thread has to execute a block:

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

    int howmany=omp_get_max_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;
}
```

Code5: Jacobi block solver-omp.c with the improvement.

After compiling and executing the binary with *Extrae* by submitting the `submit-omp-i.sh` script to boada we the following traces:



Picture3: Traces in Jacobi solver after the howmany improvement. Left: original. Right: zoom.

In this new version the distribution of work is balanced, as we can see in the second picture of the trace (zoomed 4 times to the centre of the trace). So, now we do not have any load balancing problem. Besides the efficiency has improved, we can see it in the output:

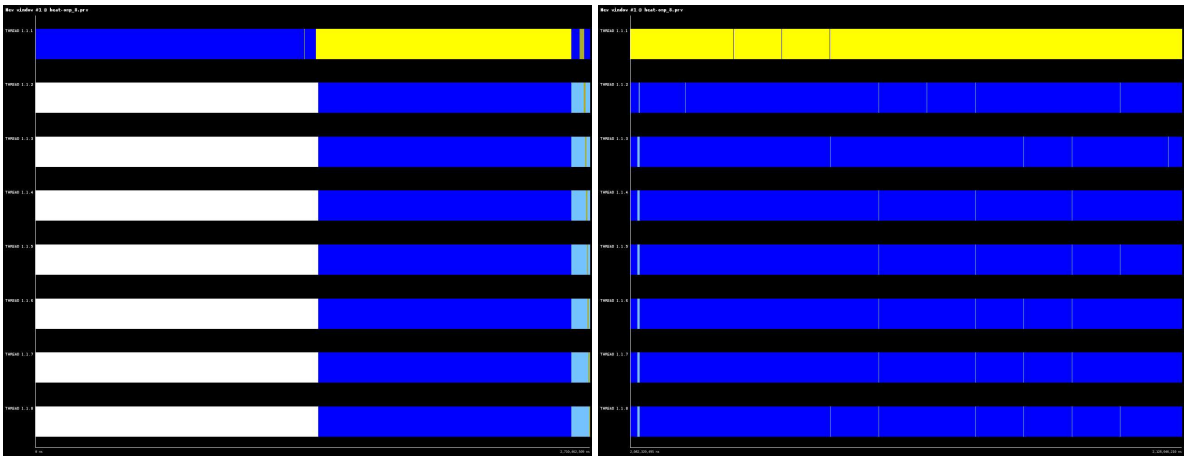
```
Iterations        : 25000
Resolution        : 254
Algorithm         : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 2.917
Flops and Flops per second: (11.182 GFlop => 3832.75 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

However, we still have a serious serialization in our parallel execution, as we can see in the second trace, where all the threats have little interruptions of work during the tasks.
This is caused because of the `aux` output is saved in the original matrix and `copy_mat()` of solver-omp.c copy one matrix to the other.
To solve the problem, we put a `#pragma omp parallel for` in the function:

```
/*
 * Function to copy one matrix into another
 */
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

Code6: Copy matrix block solver-omp.c

Then, we compile and execute it again, obtaining the following traces:



Picture4: Traces in Jacobi solver after the copy_mat improvement. Left: original. Right: zoom.

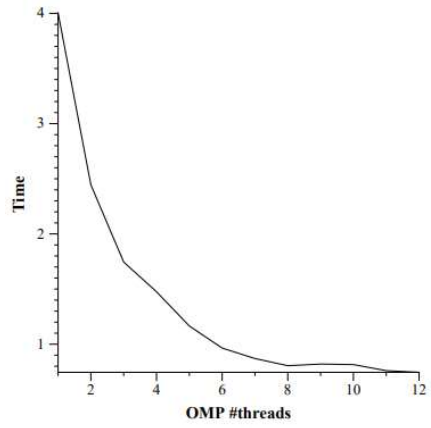
As we can see in the second picture of the trace (zoomed 4 times to the centre of the trace), there is a great improvement, due to avoiding the serialization.
Besides in this new version the distribution of work still balanced, due to the work of the `copy_mat()` is distributed equally. We also see a great improvement in the output data:

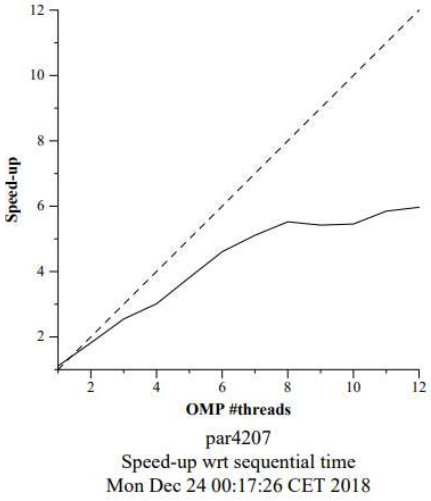
```
Iterations        : 25000
Resolution        : 254
Algorithm         : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 0.874
Flops and Flops per second: (11.182 GFlop => 12795.00 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

In each improvement, it is verified with `diff` command that the same image is formed as in sequential.

```
par4207@boada-1:~/lab5/S1$ diff heat-jacobi.ppm heat.ppm
```

Finally, we use the submit-strong-omp.sh script to queue the execution of heat-omp, obtaining the following graphics of scalability of the parallelization for different number of processors (1 to 12):





Graphic1: Jacobi heat-omp-strong.c

As we can see, the execution time is reduced in every threat is put, with a speed-up with an almost linear increase.
However, from 8 threads onwards the execution time and speed-up stabilize and the increase is reduced, this is because the parallelized part is so fast that in the total time the differences are increasingly smaller.

Session 3: Parallelization of Gauss-Seidel with OpenMP `ordered`

At this session we have parallelized the Gauss-Seidel solver using `#pragma omp for` and its `ordered` clause.

As we could see in the part of Tareador in Gauss algorithm we have dependencies in the matrix. Thus, for this part, we synchronize the parallel execution of the rows assigned to each processor in order to guarantee the dependencies.
As you can see in the code below, we put a `#pragma omp parallel for ordered(2) private(unew,diff) reduction(+:sum)` clause before the for's clauses, that creates a task for each 2-for loop, make `unew` and `diff` variables private and put a reduction when a value is added to `sum` variable. The second for loop is the one that we added to divide the columns in block and obtain a blocking per column.
Furthermore, we put a `#pragma omp ordered depend(sink: blockid-1, blockid2)` clause before the third for, that guarantees the dependencies, with the previous group (`blockid-1`) .
Finally, we added a `#pragma omp ordered depend(source)` after the third for.

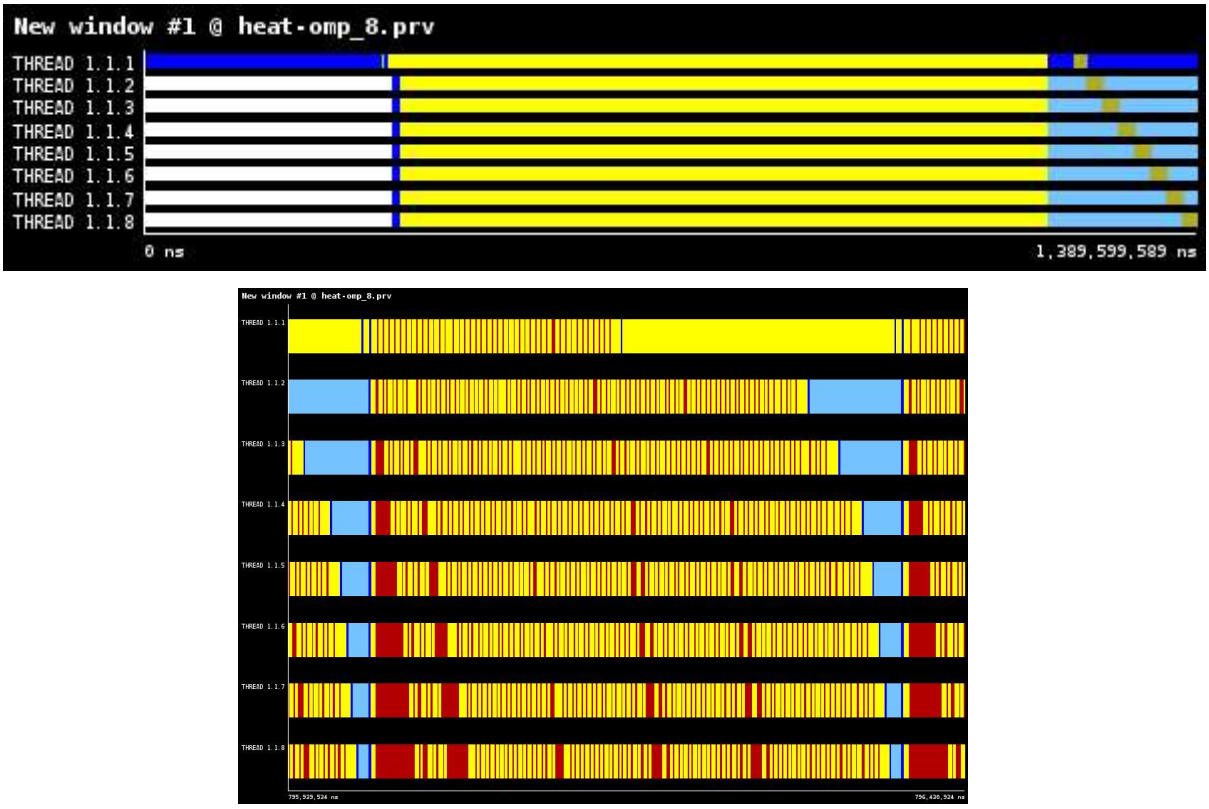
```
/*
 * Blocked Gauss-Seidel solver
 */
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany = omp_get_max_threads();
    int howmany2 = 42;

    #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
    for (int blockid = 0; blockid < howmany; ++blockid) {
      for (int blockid2 = 0; blockid2 < howmany2; blockid2++) {
        int i_start = lowerb(blockid, howmany, sizex);
        int i_end = upperb(blockid, howmany, sizex);

        int j_start = lowerb(blockid2, howmany2, sizey);
        int j_end = upperb(blockid2, howmany2, sizey);

        #pragma omp ordered depend(sink: blockid-1, blockid2)
        for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<= min(sizey-2, j_end); j++) {
            unew= 0.25 * ( u[ i*sizey   + (j-1) ]+  // left
                u[ i*sizey  + (j+1) ]+  // right
                u[ (i-1)*sizey  + j     ]+  // top
                u[ (i+1)*sizey  + j     ]); // bottom
            diff = unew - u[i*sizey+ j];
            sum += diff * diff;
            u[i*sizey+j]=unew;
          }
        }
         #pragma omp ordered depend(source)
      }
    }

    return sum;
}
```

Code7: Gauss solver function.

After that, we compiled using `make heat-omp` and submit the execution of the binary using the `submit-omp.sh` script to validate the parallelization.
We obtain the following output:

```
Iterations        : 25000
Resolution        : 254
Algorithm         : 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: 1.132
Flops and Flops per second: (8.806 GFlop => 7776.99
Convergence to residual=0.000050: 12409 iterations
```

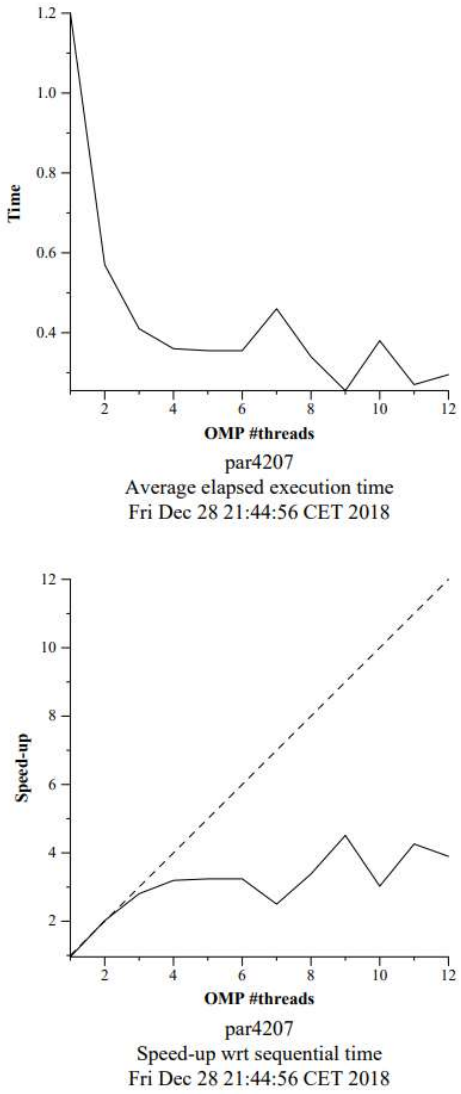
We have also verified that it works properly by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version.
Then, we instrument with Extrae by submitting the `submit-omp-i.sh` script and visualize the traces generated for the parallel execution.
We had to decrease the size of the iterations of the `test.dat` file because it could not be shown and it was not representative, obtaining the following traces:



Picture5: Traces of Gauss improvement. Up: original. Down: zoom.

As we can see in the zoom, the evolution of the paralysed part is very cyclic (because of each loop has the same structure) and depends on which part is zoomed will have assigned more to one thread than another.

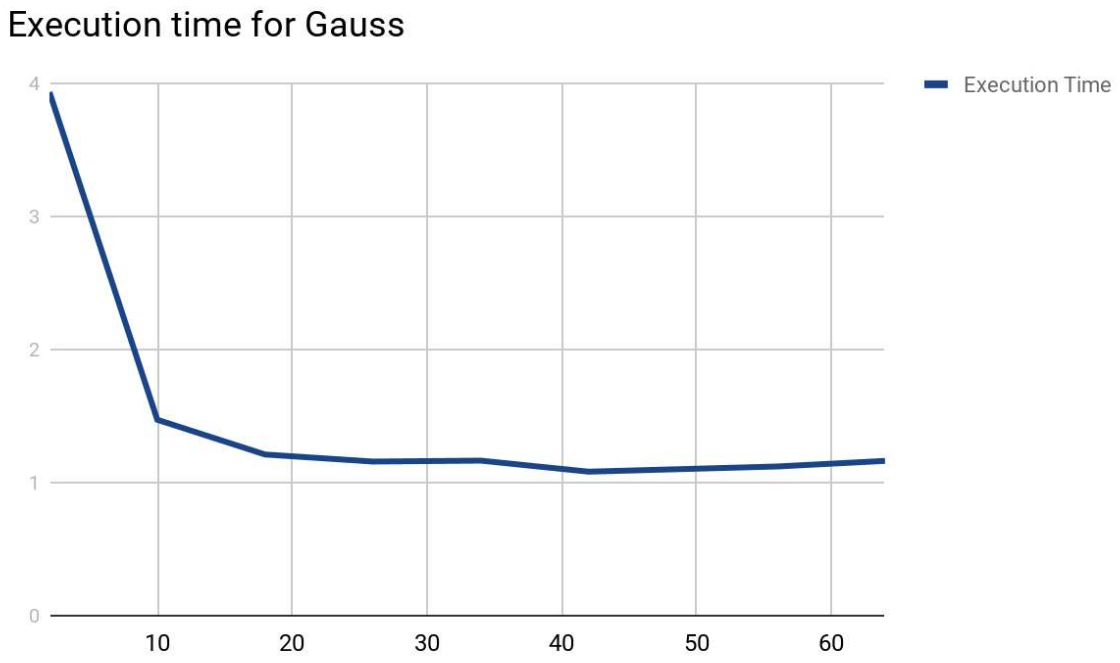
Once the parallelization is correct, we used the `submit-omp-strong.sh` script to queue the execution and we obtain the following graphics:



Graphic2: Scalibility graphics of Gauss solver.

As we can see, in general, the execution time decreases and the speed-up increase as we increase the number of threads. However, when we arrived at 4 threads, these two data are stabilized.

To obtain an optimum value for the ratio between computation and synchronization, we explore possible ratios, to see how the execution time varies, and we represent in a plot, for the execution with 8 threads:



Graphic3: Execution time vs ratio.

In our case the optimal value has been 42 since from this the time increases slightly.

Conclusions

Concluding our analysis, the Jacobi algorithm there are no load balancing or serialization problems, thus it has a good distribution of work among the threads and a block can be assigned to each thread, although to copy into the auxiliary matrix we use more resources. Instead of, in Gauss-Seidel you can not assign a block to each thread, in order to apply the parallelism, we must wait to do the previous computations, but in this case, we do not need an auxiliary matrix.
Furthermore, using Jacobi solver, we has obtained a better speed-up graphic, arriving around 6, instead of Gauss-Seidel is more irregular.