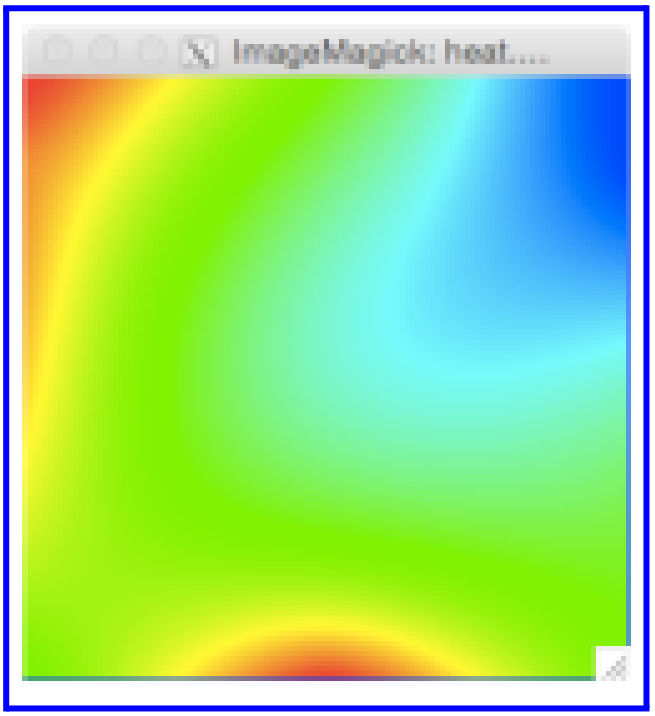
Lab 5: Geometric (data) decomposition using implicit tasks:heat diffusion equation



Adrià Redondo
Miguel Gutiérrez Jariod
Group 23
24/03/2021 - QP 2021-2022

Index:

2. Sequential heat diffusion analysis with tareador

Part I

2.1

```
par2320@boada-1:~/lab5$ make heat
icc -Wall -std=c99 -O3 heat.c solver.c misc.c -lm -o heat
par2320@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: 4.555
Flops and Flops per second: (11.182 GFlop => 2454.98 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
par2320@boada-1:~/lab5$
```

Image 2.1.1 execution of the command line ./heat test.dat -a 0 -o heat-jacobi.ppm

2.2

```
par2320@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: 6.036
Flops and Flops per second: (8.806 GFlop => 1459.07 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
par2320@boada-1:~/lab5$
```

Image 2.1.2 execution of the command line ./heat test.dat -a 1 -o heat-gauss.ppm

Part II

2.1

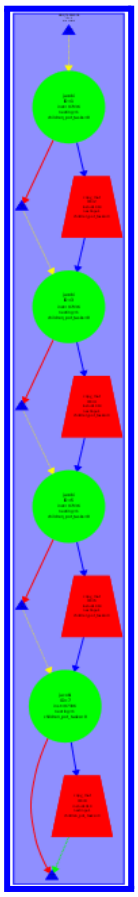


Image 2.1.1
./run-tareador.sh heat-tareador 0
0 == jacobi option
Task graph

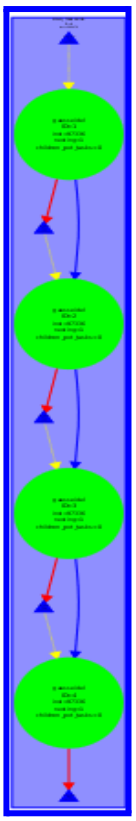


Image 2.1.2
./run-tareador.sh heat-tareador 1
1 == gauss-siedel option
Task graph

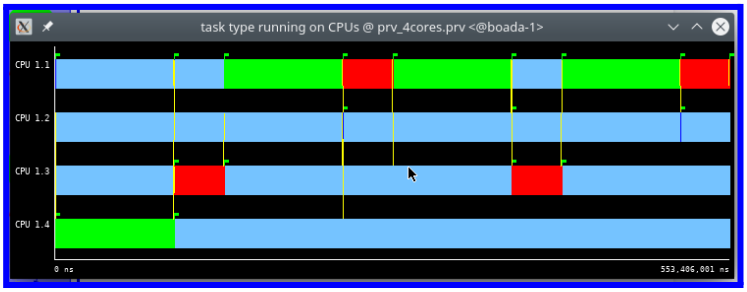


Image 2.1.3 ./run-tareador.sh heat-tareador 0, Simulation with 4 cores

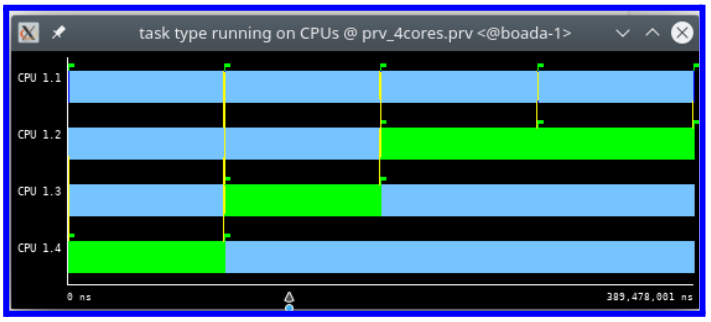


Image 2.1.4 ./run-tareador.sh heat-tareador 1, Simulation with 4 core

Question: *Is there any parallelism that can be exploited atthis granularity level?*
Answer: No, because the task graph has a heavily sequential task structure.

2.2

```
// 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("solve");
        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("solve");
      }
    }
    tareador_enable_object(&sum);

    return sum;
```

Image 2.2.2 code for the solve function with one task per block granularity

2.3

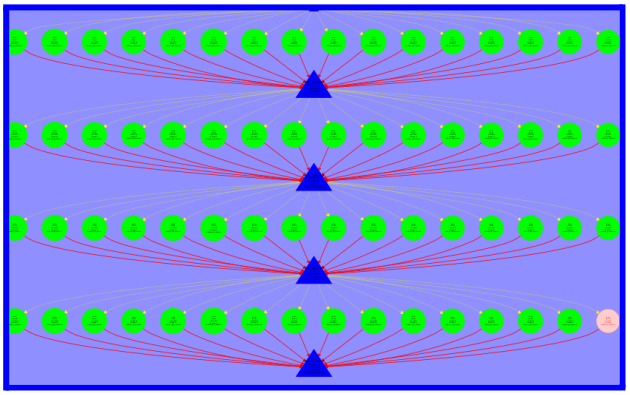


Image 2.3.1 ./run-tareador.sh heat-tareador 0, 0 == jacobi option Task graph

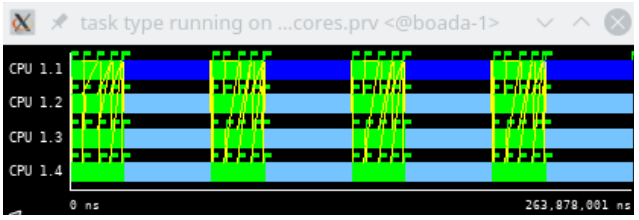


Image 2.3.2 ./run-tareador.sh heat-tareador 0, Simulation with 4 cores

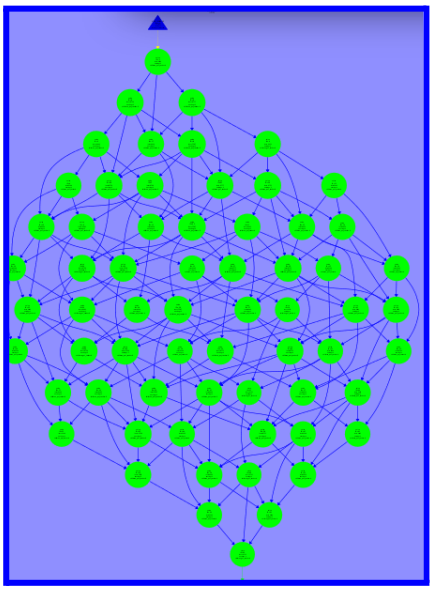


Image 2.3.3 ./run-tareador.sh heat-tareador 1, 1 == gauss-siedel option Task graph

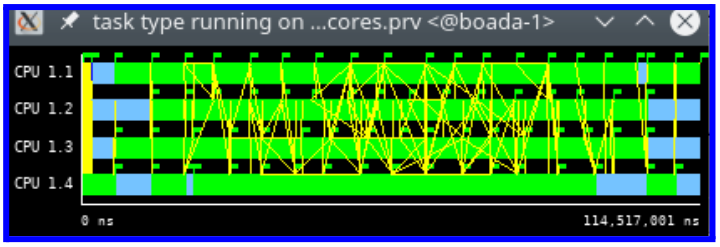


Image 2.3.4 ./run-tareador.sh heat-tareador 1, Simulation with 4 cores

We can see that the jacobi task distribution has no dependencies and is embarrassingly parallel. This is explained by the nature of the jacobi computation that uses the copy_mat function.

On the other hand, the gauss-seidel task distribution has some dependencies because this method performs the computations in one single data matrix and the order of the computations is relevant.

3. Parallelization of the heat equation solvers

Part I

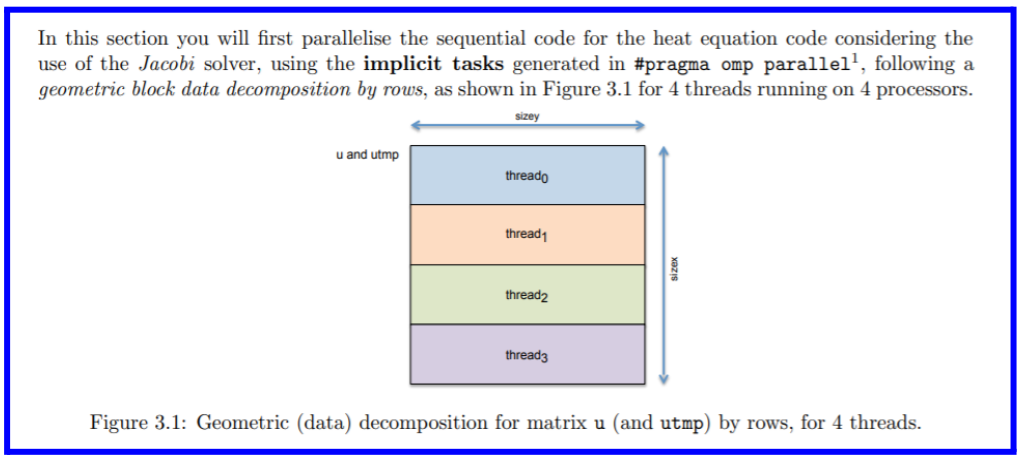

In this section you will first parallelise the sequential code for the heat equation code considering the use of the *Jacobi* solver, using the **implicit tasks** generated in `#pragma omp parallel`[1], following a *geometric block data decomposition by rows*, as shown in Figure 3.1 for 4 threads running on 4 processors.

Figure 3.1: Geometric (data) decomposition for matrix `u` (and `utmp`) by rows, for 4 threads.

Image 3.1.0 Jacobi startegy

3.1

Image 3.1.1 display of the result of the heat-omp with Jacobi strategy

```
par2311@boada-1:~/lab5$ diff heat-omp-jacobi-8-boada-2.txt heat-jacobi
-8-boada-2.txt
8,9c8,9
< Time: 3.057
< Flops and Flops per second: (11.182 GFlop => 3657.90 MFlop/s)
---
> Time: 4.531
> Flops and Flops per second: (11.182 GFlop => 2468.03 MFlop/s)
par2311@boada-1:~/lab5$
```

Image 3.1.2 diff between the jacobi strategy and the original code

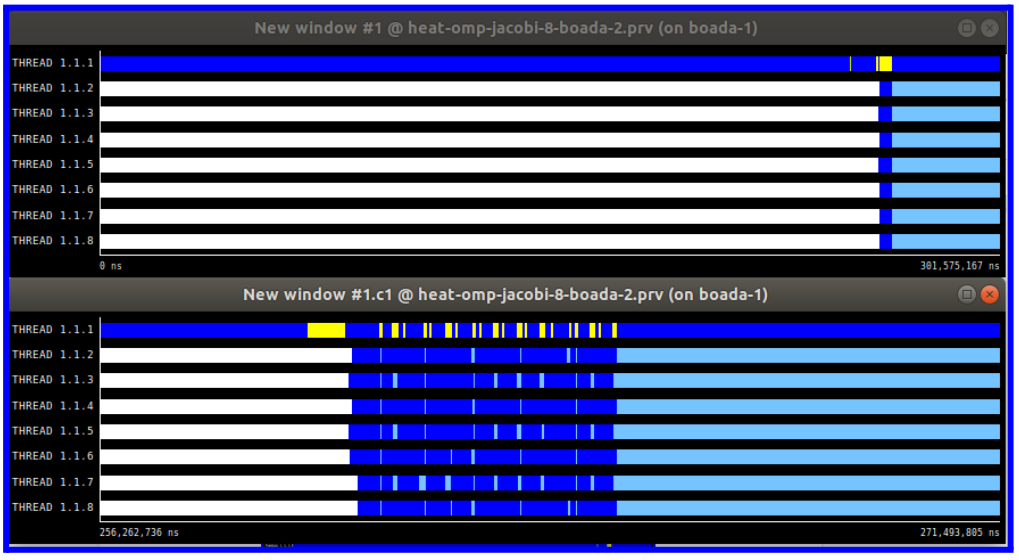


Image 3.1.3 paraver simulation of the Jacobi strategy with 8 processors

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

    //int nblocksi=8;
    //int nblocksj=1;
    #pragma omp parallel private(tmp,diff) reduction(+:sum)
    {
    int myid = omp_get_thread_num();
    int how_many = omp_get_num_threads();
      int i_start = lowerb(myid, how_many, sizex);
    int i_end = upperb(myid, how_many, sizex);
    for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
      for (int j=1; j<=sizey-2; 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;
}
```

Image 3.1.4 code of the Jacobi strategy

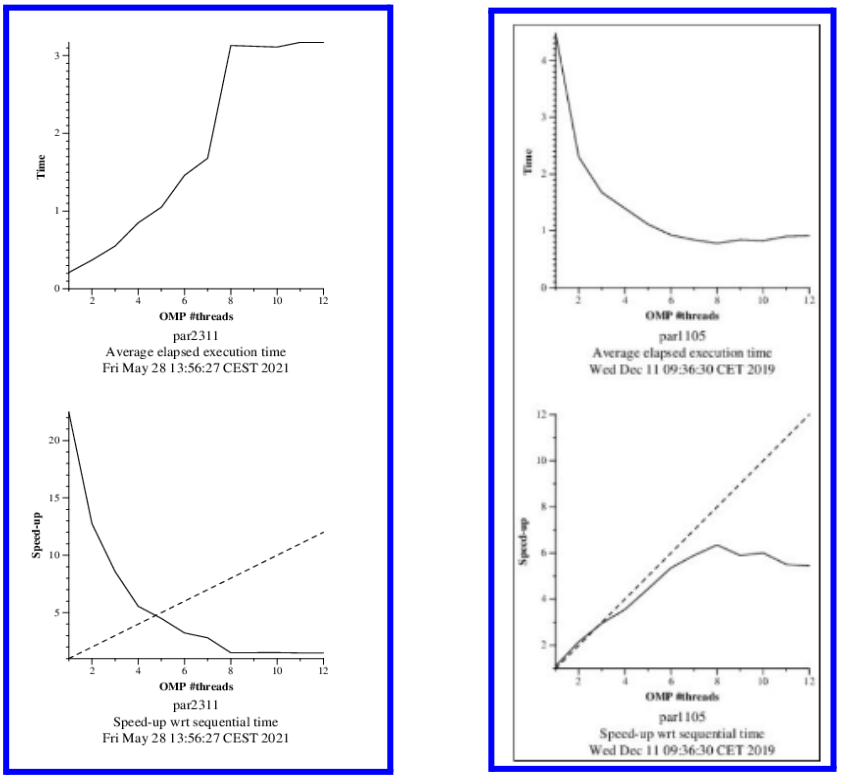


Image 3.1.5 plot of the Jacobi strategy (left: our simulation vs right: the correct one)

In this part of the practice we have developed the Jacobi strategy. As we can see in image 3.1.0 we divided the heat matrix into blocks where they take all the longitude of the x axis and for the y axis is where the different threads will do their job.

Furthermore, in image 3.1.4, we can appreciate the code of the strategy that we have developed to battle the problem. As the practice demands only implicit tasks we have opened the parallel region and later on we have obtained the id of each thread. Once it's done we begin our route of dividing in blocks for each thread that we have, so each of it treats the corresponding block. Finally we have added the private(tmp,diff), so each threat takes their value without modifying the others. Also we had the reduction for the variable sum so for each thread obtain the global sum without having problems of data race.

Later on we have obtained the heat image, equal as the original, as we can see on image 3.1.1. Then in image 3.1.2 we can appreciate the difference between the jacobi strategy and the original code, the first one the execution is faster and also it takes more Mflops/s compared to the original one.

On the image 3.1.5 we can see the plot that has been generated by our code vs the one that we think that should be the correct answer because as increasing the number of threads the time should be reduced and in the speed up case as having more threads it should increase. We have tried so many times to obtain the main result but we can find the error that keeps generating wrong answers.

Finally in image 3.1.3 we have executed the paraver simulation. We can appreciate the parallelization by blocks in which each thread develops their jobs that correspond. As having parallelized also the copy_mat we can expect a better results as the picture shows so their can be running for the most of the time.

Part II

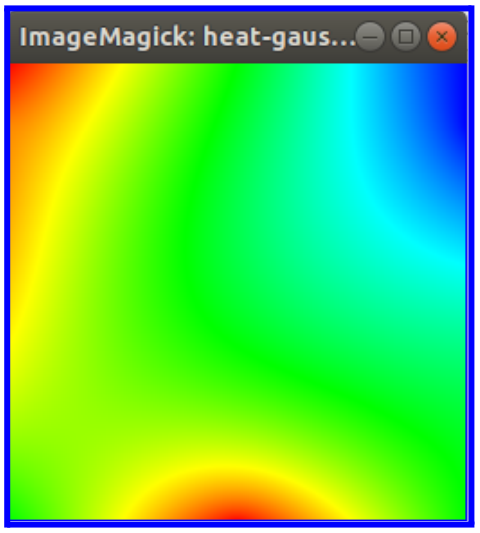


Image 3.2.1 display of the result of the heat-omp with Gauss strategy

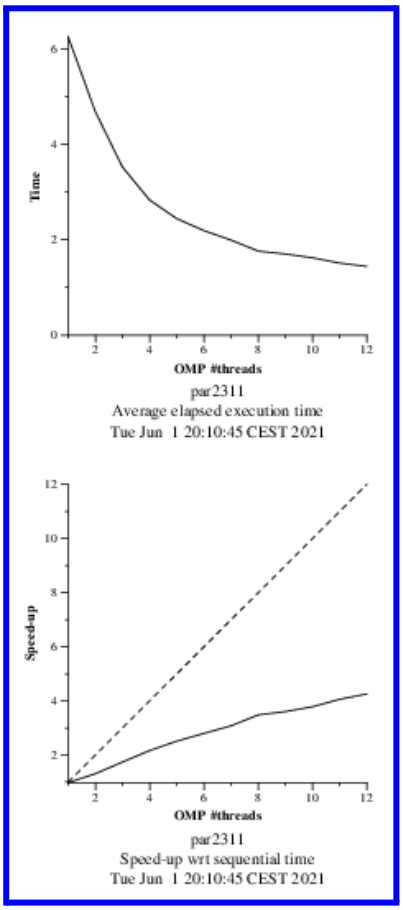


Image 3.2.2 plot of the Gauss strategy with the code expeculated

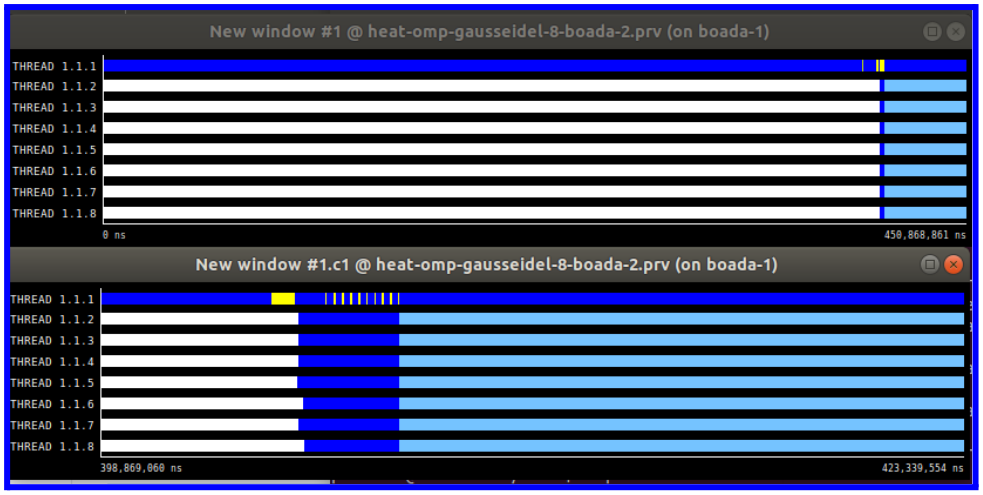
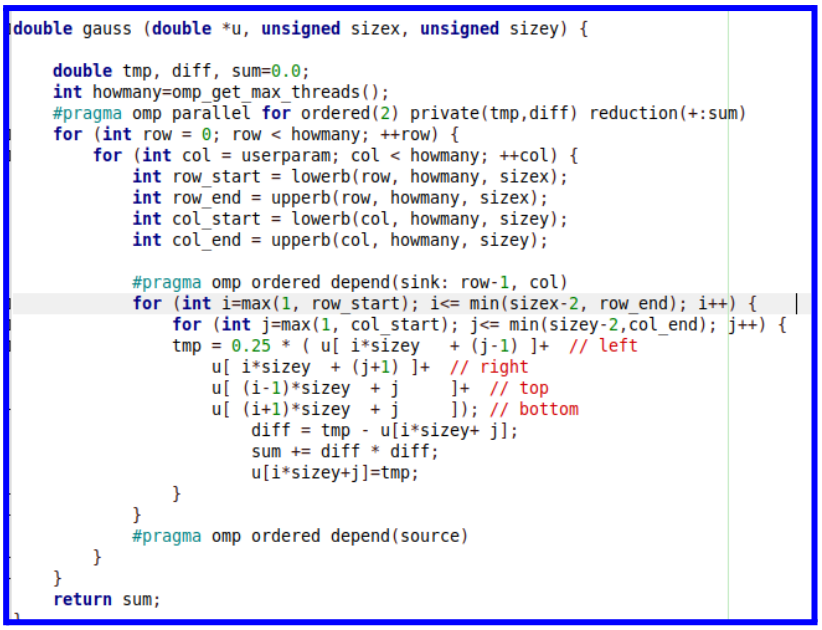


Image 3.2.3 paraver simulation of the Gauss strategy with 8 processors

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

#pragma omp parallel num_threads(P)
  {
    int myid = omp_get_thread_num();

  //if (u == unew) whichone = 0;
  //else whichone = 1;
  while (myid < P){

      int tmp;
      do {
          #pragma omp atomic read
          tmp = next[myid-1];
      } while (tmp == 0);


      int how_many = omp_parallel_num_threads();
      #pragma omp parallel private(tmp,diff) reduction(+:sum)
      for (int blocki=0; blocki<how_many; ++blocki) {
        int i_start = lowerb(blocki, how_many, sizex);
        int i_end = upperb(blocki, how_many, sizex);
        for (int blockj=0; blockj<how_many; ++blockj) {
          int j_start = lowerb(blockj, how_many, sizey);
          int j_end = upperb(blockj, how_many, 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;
                  u[i*sizey+j] = tmp;
              }
            }
          }
      }
       if (myid < P-1){
          #pragma omp atomic write
          next[myid] = 1;
      }
  }
```

Image 3.2.4 our code of the Gauss strategy (failed)



```
double gauss (double *u, unsigned sizex, unsigned sizey) {

    double tmp, diff, sum=0.0;
    int howmany=omp_get_max_threads();
    #pragma omp parallel for ordered(2) private(tmp,diff) reduction(+:sum)
    for (int row = 0; row < howmany; ++row) {
        for (int col = userparam; col < howmany; ++col) {
            int row_start = lowerb(row, howmany, sizex);
            int row_end = upperb(row, howmany, sizex);
            int col_start = lowerb(col, howmany, sizey);
            int col_end = upperb(col, howmany, sizey);

            #pragma omp ordered depend(sink: row-1, col)
            for (int i=max(1, row_start); i<= min(sizex-2, row_end); i++) {
                for (int j=max(1, col_start); j<= min(sizey-2,col_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;
                        u[i*sizey+j]=tmp;
                }
            }
            #pragma omp ordered depend(source)
        }
    }
    return sum;
}
```

Image 3.2.5 code of the Gauss strategy with explicit tasks to obtain at least some results

In this part of the practice we need to implement the Gauss strategy. After having tried so many times and having many problems we haven't obtained the code to take the corresponding comments to the different parts of this strategy. We don't have properly mixed the Gauss main model with the accomplishment said about the vector that could register the dependencies. We have understood that the method uses squares to the different threads so the main part is reduced into smaller ones so the threads can be distributed with coherence. In the image 3.2.4 is the version asked before.

As follows with wrong answers with our version, we obtain a code with explicit tasks so at least we tried to obtain some results to discuss. We can see it in image 3.2.5, this version has two dependencies, as we can see , so takes the proper limits for each thread on each row and column. Moreover the private tmp and diff are one of the clausules to evit that all the threads evoke with errors on each operation that update the values commented before. Also the reduction part on the sum variable to share the value to the different threads, and finally the for to take care with the values that go over the bulce and increments each time.

In image 3.2.1 we have obtained the image of the heat, as we can see it isn't as equal as the main one, but keeps the essence. As it shows also the code isn't fully correct. However in image 3.2.2 we have executed the plot of the strategy, more or less the main essence that shows is that as increasing the number of threads the time reduces so it is because of the job of the strategy that has the capacity to distribute threads properly so the parallelization accomplishes with credentials. Also for the speedup case, as the number of threads goes up because with more the main task is distributed in more parts so is divided with more threads and finishes earlier.

Finally we can see in image 3.2.3 the paraver execution for 8 processors. We can appreciate that in the timeline the execution of all the parallelization takes a little, it takes more to initialize. If we zoom in the proper space we can see that each threat takes the same time to be executed. Compared with the other strategy we think that it should be faster because of the distribution of the threads and the amount of work that they have to do.

At least we couldn't realise the last part because when trying again to add the userparam for both versions, no one makes it correctly. We have supposed that attributing the userparam to the nblockj it corresponds to the column therm. So depending on the value it can take a range more width for the shape of the square, making bigger the fragments that the main task is divided. But again as not having a reference it is a supposition.