David Martos
Mariona Farré
Grupo: paco1208
Curso: 2022-23

PACO:
Laboratorio 5

Index:

1. Before starting this laboratory assignment

2. Sequential heat diffusion program and analysis with Tareador 5

Ejecutar con los solucionadores dados Jacobi y Gauss-Seidel ver su tiempo de ejecución y ver su imagen:

JACOBI:
./heat test.dat -a 0 -o heat-jacobi_SEQ.ppm

display heat-jacobi_SEQ.ppm

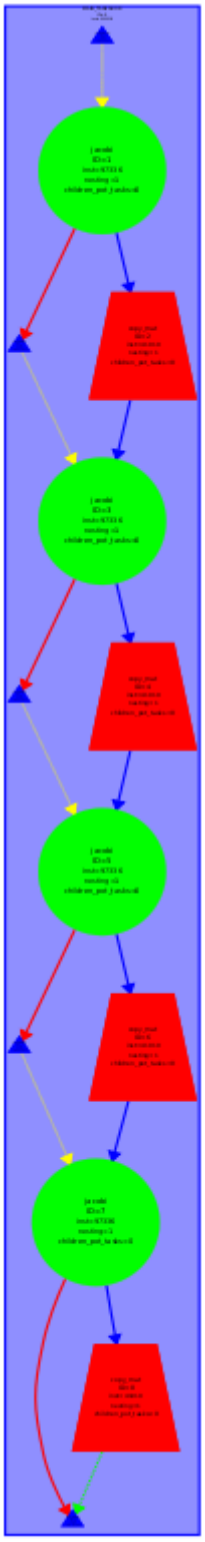
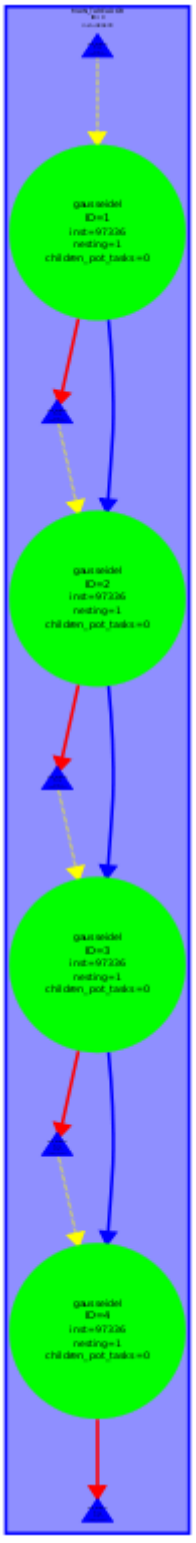
GAUSS:

./heat test.dat -a 1 -o heat-gauss_SEQ.ppm

display heat-gauss_SEQ.ppm

For the deliverable: Include the task dependency graph shown by Tareador. Is there any parallelism that can be exploited at this granularity level?
Para ver si los dos códigos se pueden paralelizar más para obtener una granularidad más fina, debemos ver el gráfico de dependencias original, ver qué tascas tienen dependencias de entrada y salida, y después ver si se puede mejorar creando una nueva tarea dentro del segundo bucle.

Gráficos originales:

JACOBI:	GAUSS:
	

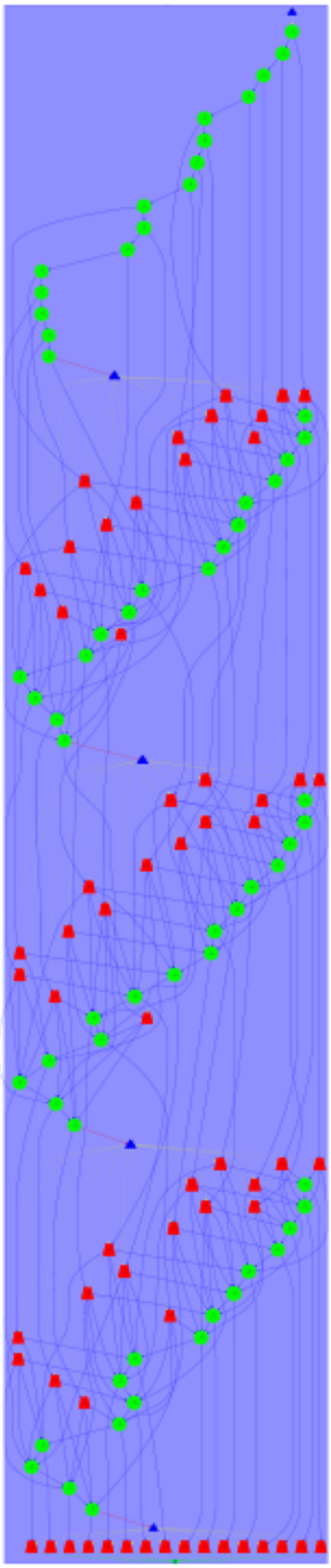
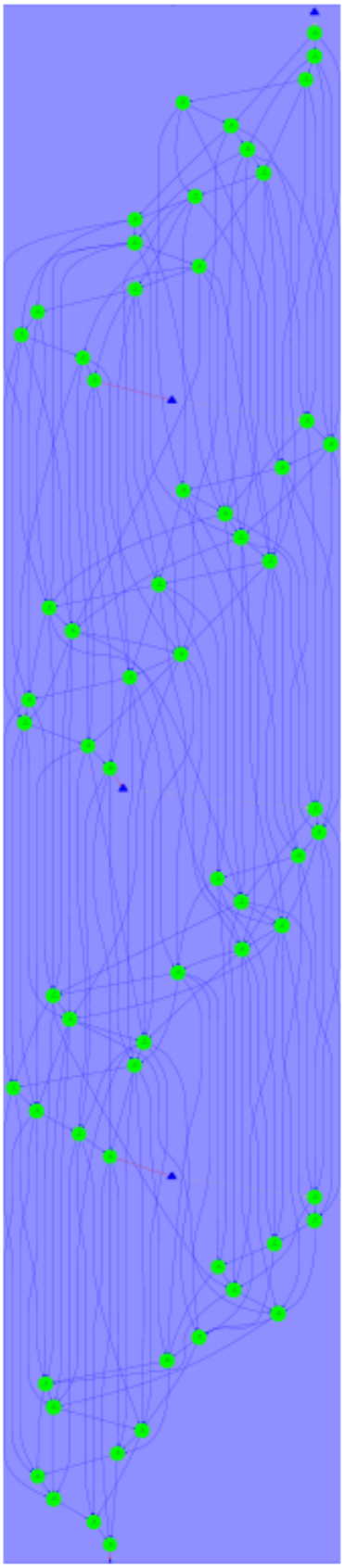
For the deliverable: Include the excerpt of the code that you have modify in order to specify one task per block.

Poner una tasca por bloque:

GNU nano 6.2 solver-tareador.c

```
extern int userparam;

// 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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
      tareador_start_task("jacob");
       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("jacob");
      }
    }
}

// 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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("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("block");
      }
    }
        //tareador_enable_object(&sum);


paco1208@boada-7:~/lab5$
```

Gráficos de tareas con el código inicialmente modificado:

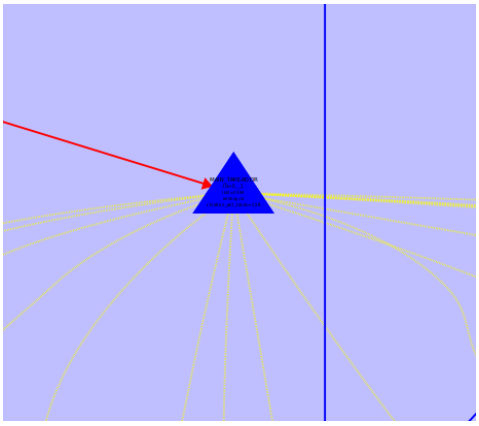
JACOBI:	GAUSS:
	

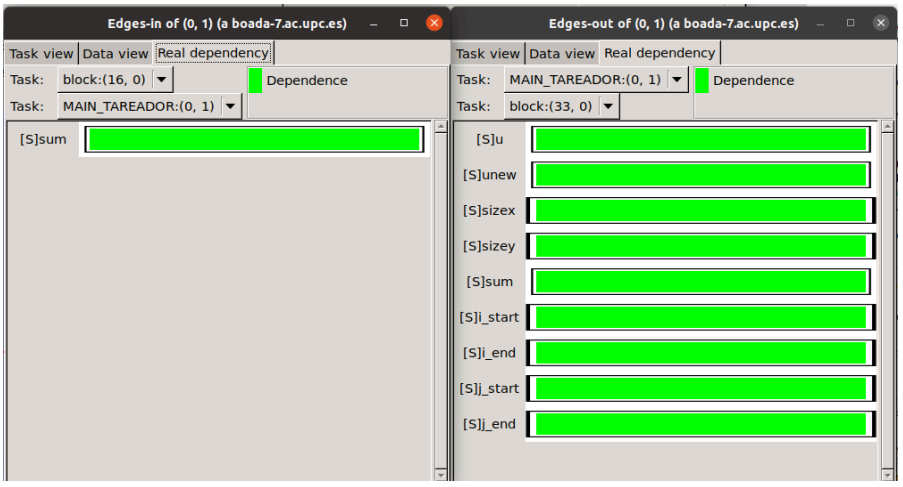
a) Which variable is causing the serialization of all the tasks? Use the **Dataview** option in Tareador to identify it

.

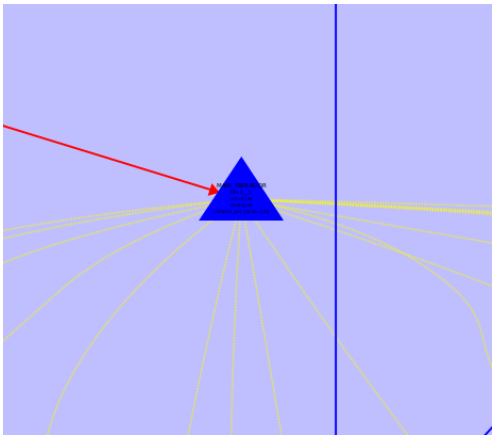
Ver que tareas bloquean y crean serialización.

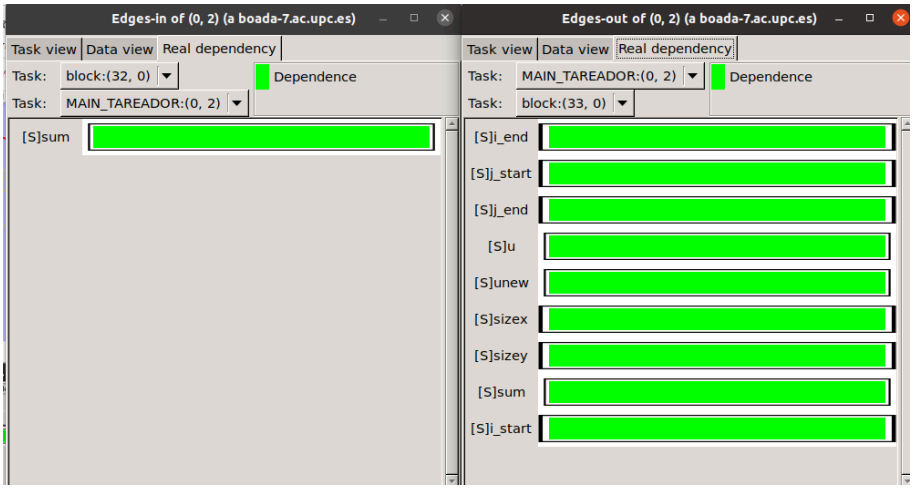
JACOBI:





GAUSS:

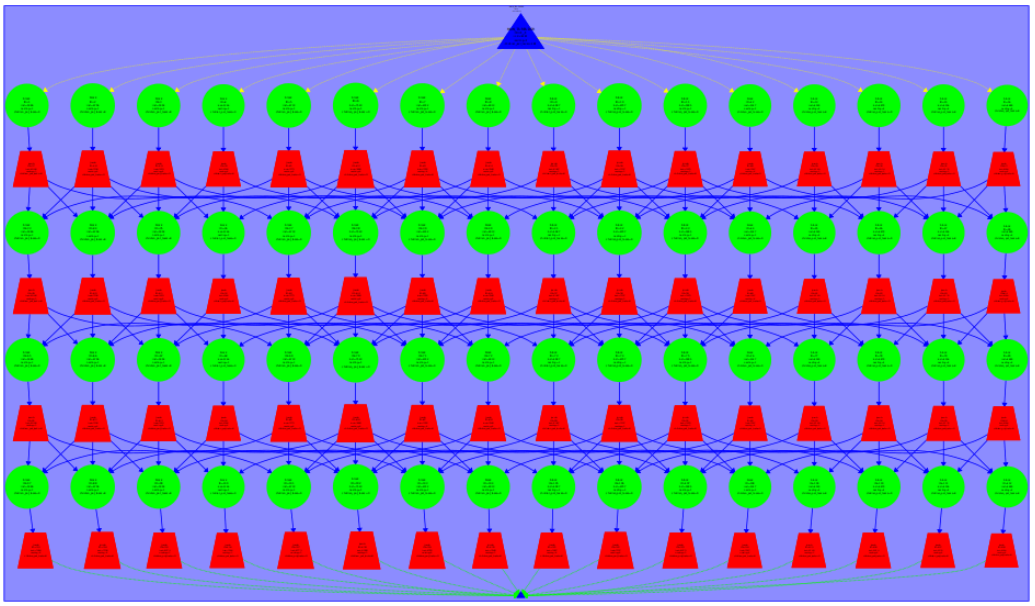
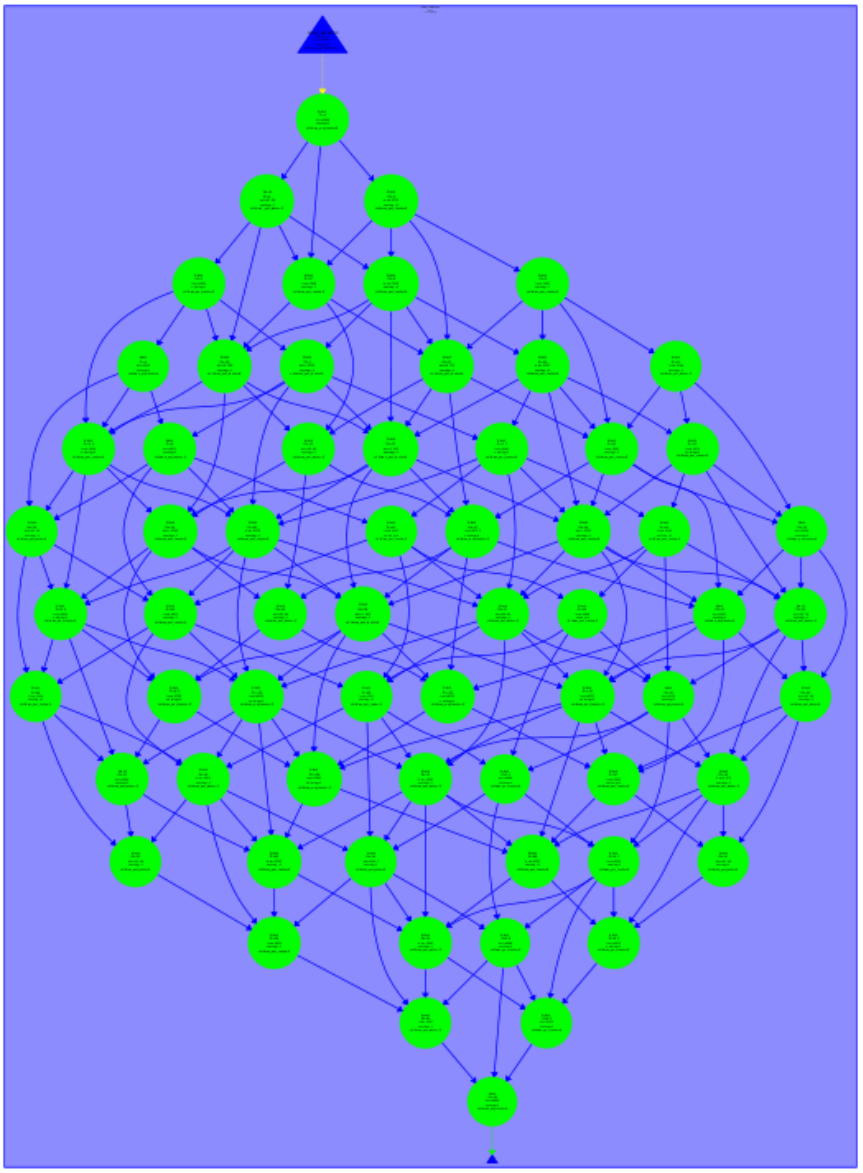




b) In order to emulate the effect of protecting the dependences caused by this variable, you can use the tareador disable object and tareador enable object calls, already introduced in the code as comments. With these calls you are telling to Tareador to filter the dependences caused by the variable indicated as object. Uncomment them, recompile and execute.

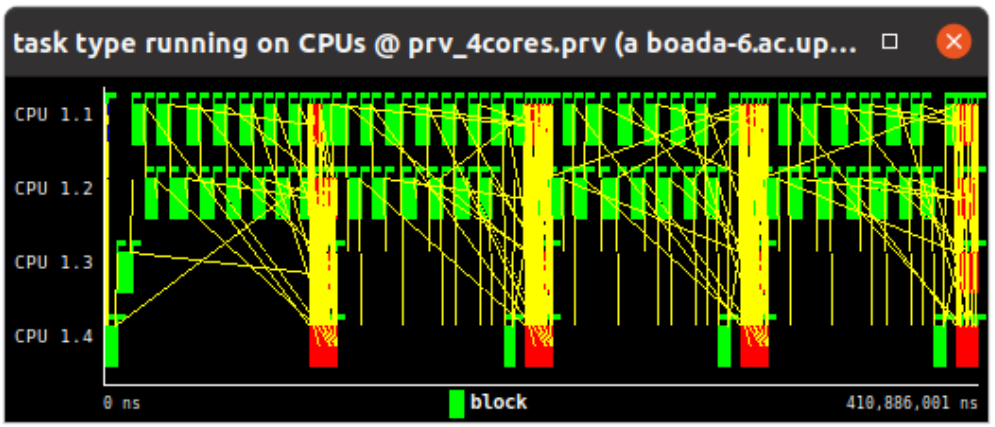
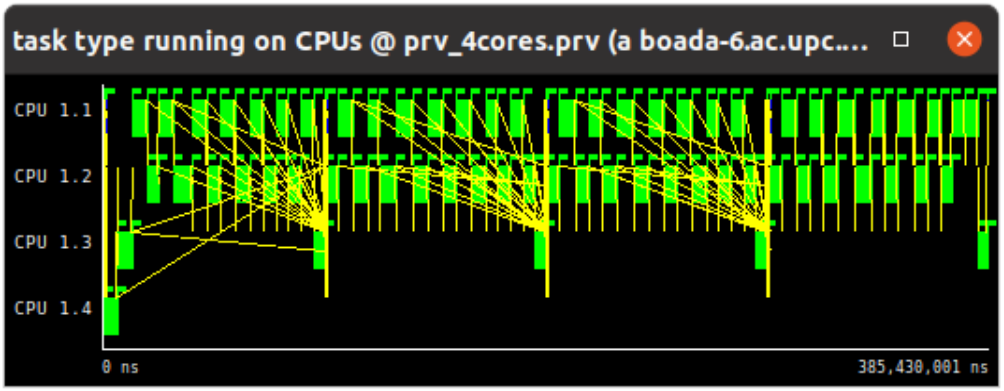
```
GNU nano 6.2                                heat-tareador.c
            fprintf(stderr, "Error in Solver initialization.\n\n");
            usage(argv[0]);
        return EXIT_FAILURE;
        }

    // full size (param.resolution are only the inner points)
    np = param.resolution + 2;

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

    iter = 0;
    while(1) {
        switch( param.algorithm ) {
            case 0: // JACOBI
            //tareador_start_task("jacobi");
            residual = solve(param.u, param.uhelp, np, np);
            //tareador_end_task("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-SEIDEL
            //tareador_start_task("gausseidel");
                    residual = solve(param.u, param.u, np, np);
            //tareador_end_task("gausseidel");
                    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;
    }

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

    fprintf(stdout, "Time: %04.3f \n", runtime);
    fprintf(stdout, "Flops and Flops per second: (%3.3f GFlop => %6.2f MFlop/s)\n",
            flop/1000000000.0,
            flop/runtime/1000000);
    fprintf(stdout, "Convergence to residual=%f: %d iterations\n", residual, iter);
```

```
  GNU nano 6.2                                solver-tareador.c
// 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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
      tareador_start_task("jacob");
       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("jacob");
      }
    }
}

// 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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("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("block");
      }
    }
tareador_enable_object(&sum);

    return sum;
}
```

JACOBI	
GAUSS	

For the deliverable: Which variable was causing the serialisation of all the tasks? Are you obtaining more parallelism? How will you protect the access to this variable in your OpenMP implementation? Simulate the execution when using 4 threads and extract your conclusions. Is there any other part of the code that can also be parallelised?. If so, modify again the instrumentation to parallelise it

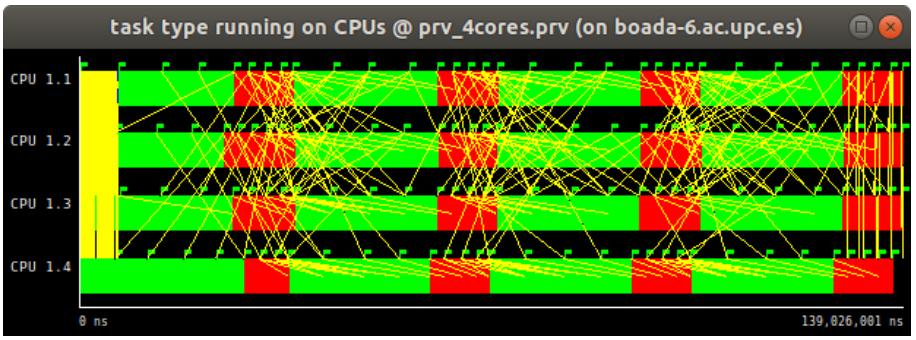
Podemos ver que en los gráficos anteriores, la roja es la tarea que hemos modificado y en verde la tarea del solver ya sea jacobi o gauss que también se modifica, era la tarea en azul la que causaba todos los problemas de serialización.

JACOBI	
GAUSS	

Con la ejecución con threads vemos que el verde se paraleliza correctamente, però la funcion de copy-math, que es el hilo amarillo, se pude paraelizar para que sea mas eficiente.

```
// 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) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
      tareador_start_task("copy_mat");
       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_start_task("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=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;
}
```

Ejemplo de el heat Jacobi, ha que es el solver que usa el copy mat:



3. Parallelisation of the heat equation solvers

3.1. Jacobi Solver

3.1.1. First Implementation

implicit tasks generated #pragma omp parallel - geometric bloc data decomp by rows parallelitation function solve
solver-omp.c
complete parallel code jacobi solver
compile: make heat-omp
submit: sbatch submit-omp.sh heat-omp 0 1
diff file generated original seq version

For the deliverable: Is the execution time of the OpenMP version of heat using 8 threads reduced compare to the sequential execution or execution using 1 thread? if not, you should reconsider your implementaton. For instance, what kind of synchronization are you using? Review different strategies to avoid and/or reduce the amount of synchronization overheads per interation. Include an excerpt of the code to show the OpenMP annotations you have added to the code.

Se Código original:

```
// 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 // 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;
}
```

Código con paralelización:

```
// 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 private(diff,tmp) 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;
```

Con 1 thread:

```
paco1208@boada-7:~/lab5$ diff heat-jacobi.ppm heat-jacobi_SEQ.ppm
paco1208@boada-7:~/lab5$
```

Con 8 threads:

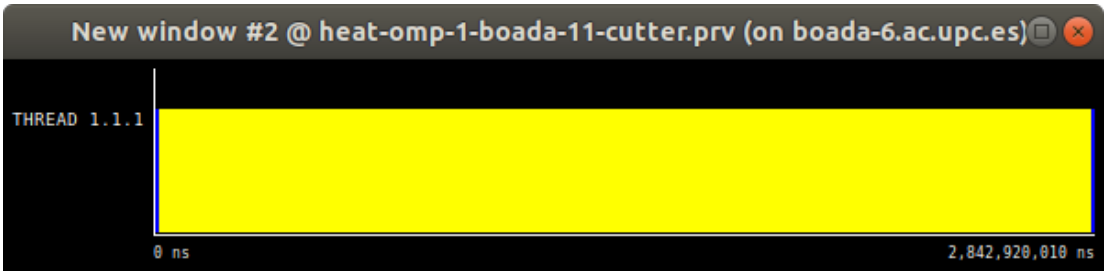
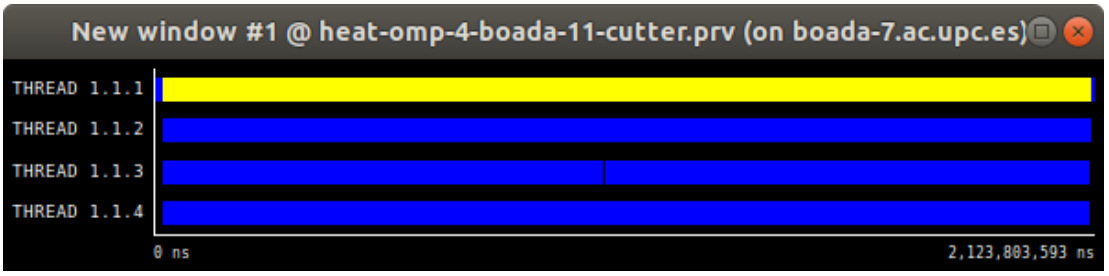
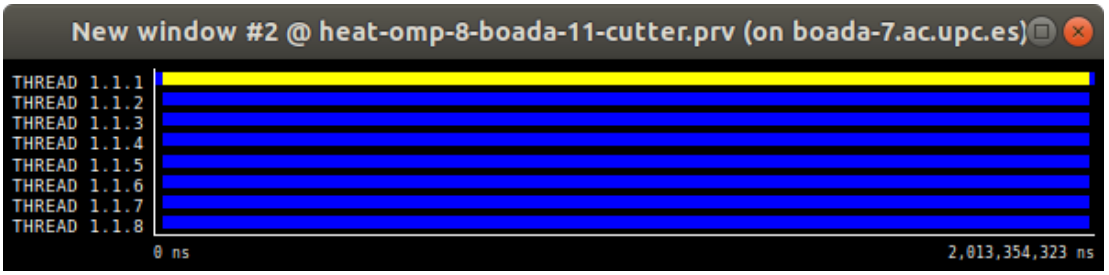
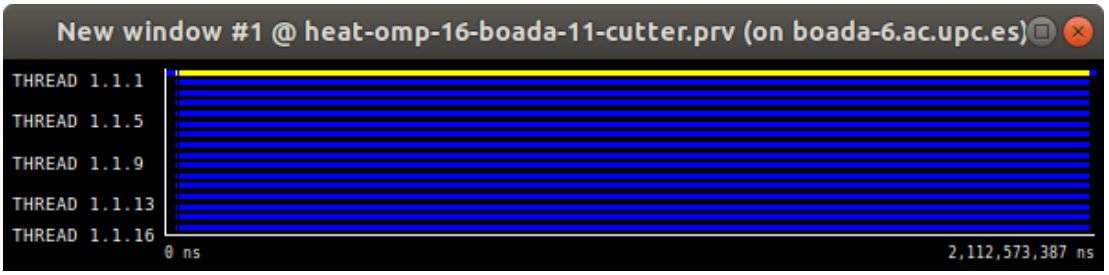
```
paco1208@boada-6:~/lab5$ diff heat-jacobi.ppm heat-jacobi_SEQ.ppm
paco1208@boada-6:~/lab5$
```

Vemos que con 8 threads no da errores comparándolo con el jacobi sequencial diciendo que los dos programas tienen el mismo resultado y esta paralización es correcta.

3.1.2. Overall Analysis

For the deliverable: Include the Modelfactor tables. Is the scalability that is obtained with this initial parallelisation appropriate? Which is the metric reported by modelfactors.py that you should address first?

Igualmente que el tiempo mejore, se debe paralizar la función que falta, que sería la función de copy-mat para poder mejorar el código.

1	
4	
8	
16	

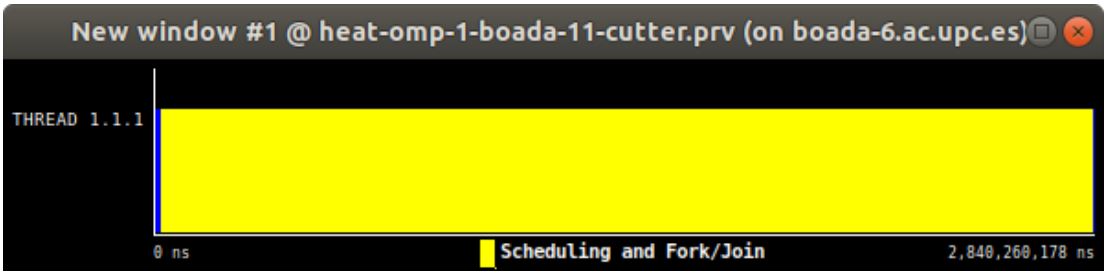
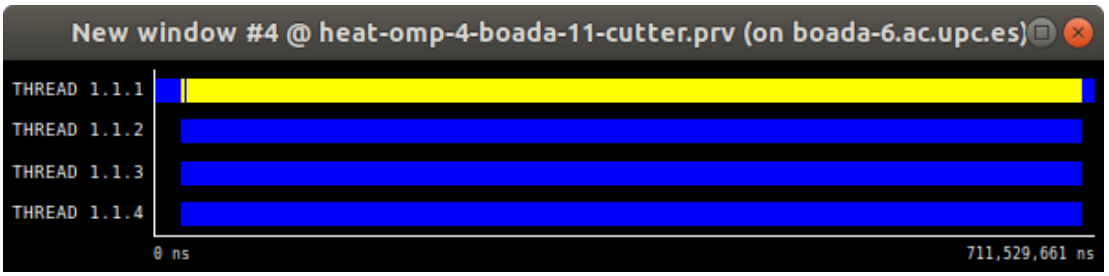
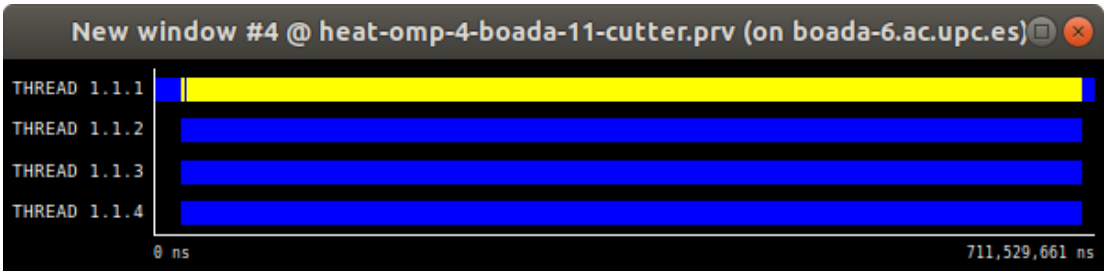
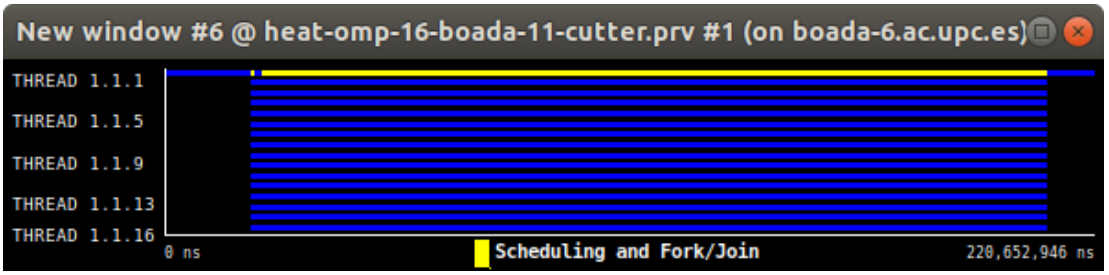
3.1.3. Detailed Analysis

We recommend you to open with paraver the trace that has been generated for 16 threads and observe what is causing the low value for that metric.
For the deliverable: Include the window timelines or paraver Hints that you consider necessary. What is the region of the code that is provoking the low value for the parallel fraction in your parallelisation?

3.1.4. Optimization

Parallelise other parts of the code in order to improve the efficiency of your parallel code. Compile the new version and submit its execution to the queue using the submit-omp.sh script, specifying the binary file, the use of the Jacobi solver and 16 threads. Make sure the new parallel version still generates correct results.
For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code.
Se puede paralelizar la función copy_mat para que sea mejor su eficiencia, para eso se debe poner un #pragma omp parallel para paralizar la siguiente zona

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

    int nblocksi=8;
    int nblocksj=1;
#pragma omp parallel
    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);
        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 private(diff,tmp) 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;
}
```

1	
4	
8	
16	

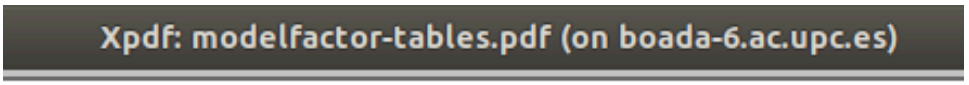
JACOBI:

```
paco1208@boada-6:~/lab5$ ./heat-omp test.dat -a 0
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: 2.251
Flops and Flops per second: (11.182 GFlop => 4967.89 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
paco1208@boada-6:~/lab5$
```

GAUSS:

```
paco1208@boada-6:~/lab5$ ./heat-omp test.dat -a 1
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: 4.439
Flops and Flops per second: (8.806 GFlop => 1983.74 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
paco1208@boada-6:~/lab5$
```

Tabla de modelfactors creada con el código con copy_mat también paralelizado:



Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.84	0.71	0.45	0.22
Speedup	1.00	3.99	6.27	12.87
Efficiency	1.00	1.00	0.78	0.80

Table 1: Analysis done on Thu Dec 29 08:49:31 PM CET 2022, paco1208

3.2. Gauss-Seidel solver:

3.2.1. First Implementation:

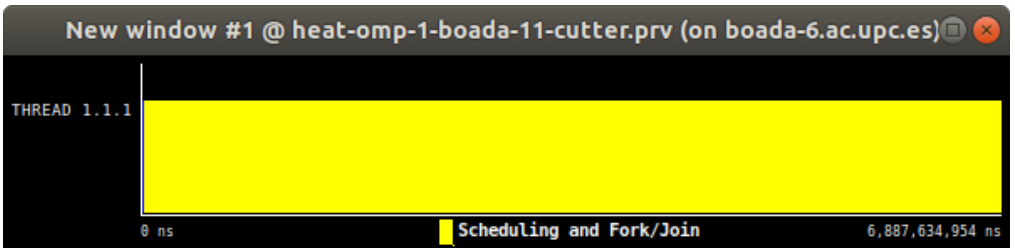
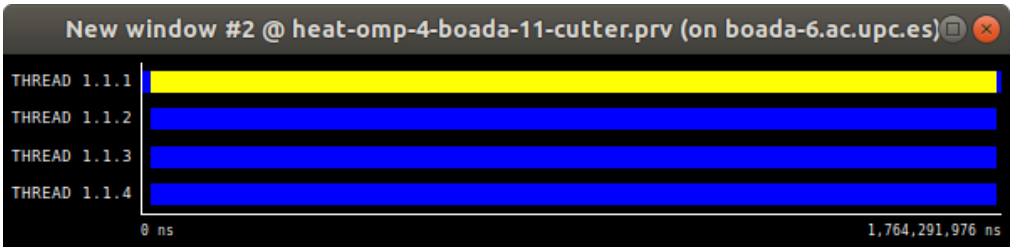
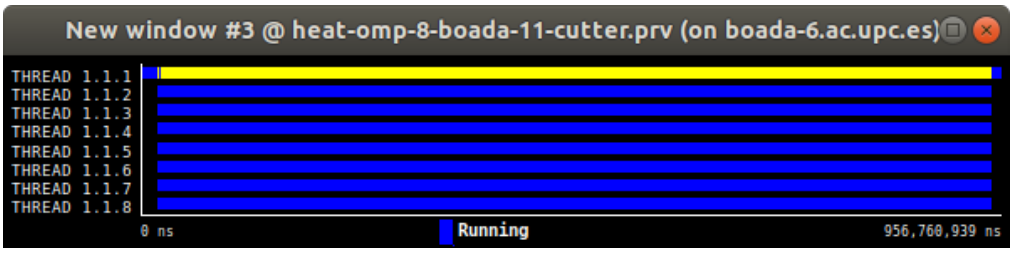
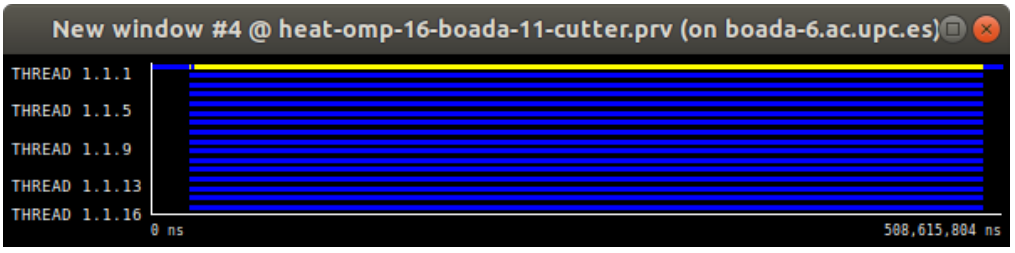
Implementar en la función solve las modificaciones tal y como la documentación nos dice

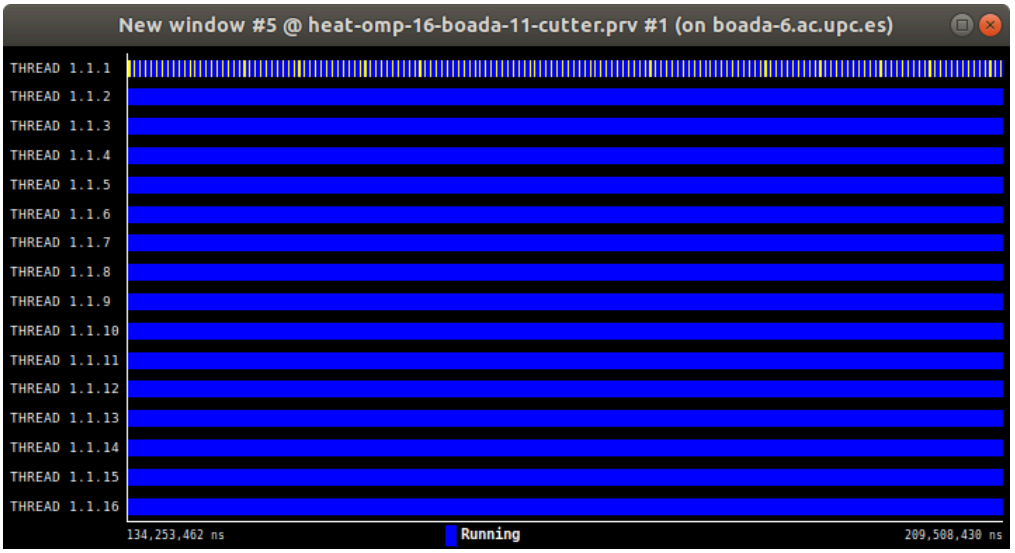
```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    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(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=nblocksi;
    int m[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
#pragma omp parallel private(diff,tmp) reduction(+:sum)
// complete data sharing constructs here
    {
        int cont;
    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);
      if ((u == unew) && blocki != 0) {
        while(cont<=blockj) {
          #pragma omp atomic read
          cont = m[blocki-1];
        }
      }
      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 (u == unew) {
        #pragma omp atomic write
        m[blocki] = m[blocki] + 1;
        }
      }
    }
```

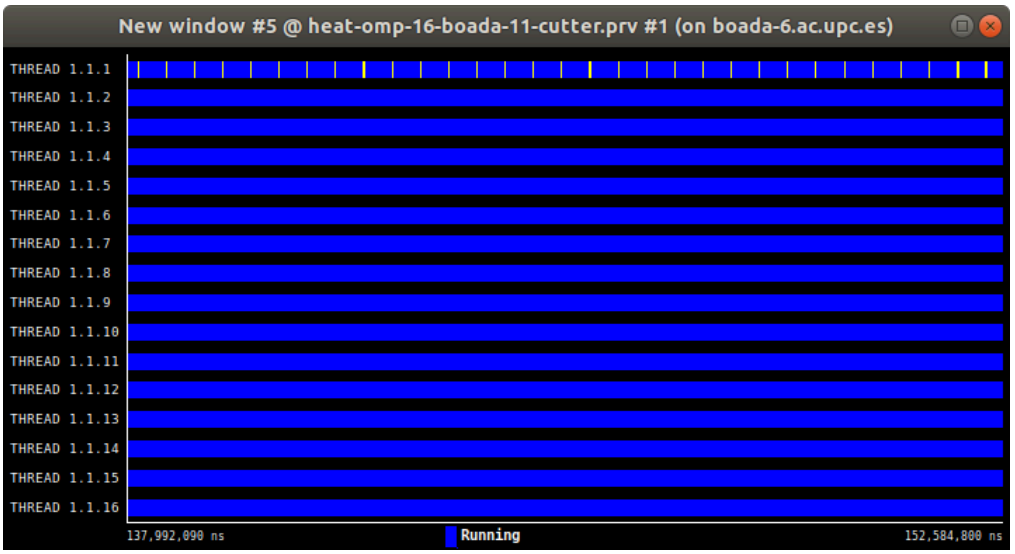
Al ejecutarse y ser comparado con el resultado sequen cuál de gauss no da ninguna diferencia en la solución, sabiendo que el código modificado continua creando una buena solución:

```
paco1208@boada-6:~/lab5$ diff heat-gauss.ppm heat-gauss_SEQ.ppm
paco1208@boada-6:~/lab5$
```

3.2.2. Overall analysis

1	
4	
8	
16	


New window #5 @ heat-omp-16-boada-11-cutter.prv #1 (on boada-6.ac.upc.es)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
THREAD 1.1.11
THREAD 1.1.12
THREAD 1.1.13
THREAD 1.1.14
THREAD 1.1.15
THREAD 1.1.16
134,253,462 ns
Running
209,508,430 ns


New window #5 @ heat-omp-16-boada-11-cutter.prv #1 (on boada-6.ac.upc.es)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
THREAD 1.1.11
THREAD 1.1.12
THREAD 1.1.13
THREAD 1.1.14
THREAD 1.1.15
THREAD 1.1.16
137,992,090 ns
Running
152,584,800 ns

3.2.3. Detailed Analysis

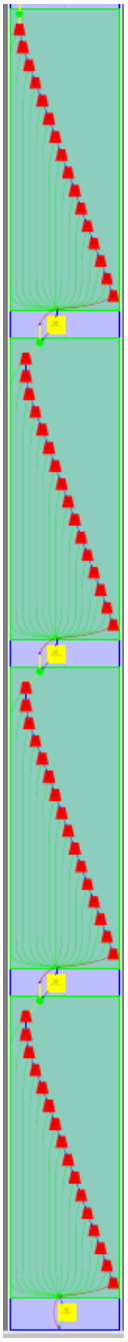
For the deliverable: Include the Modelfactor tables, the plot of scalability, and the window timelines or paraver Hints that you consider necessary. Is the scalability observed appropriate? Is there any metric reported by modelfactors.py that you should further investigate? Do you think we can increase the parallelism

Podemos observar que hay una menor escalabilidad que en la versión de Jacobi, y comparando los tiempos entre los solvers, la versión de Gauss no le va mejor la paralelización, porque no usa la función copy_mat como lo hace la versión de Jacobi.

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	6.89	1.76	0.96	0.51
Speedup	1.00	3.90	7.20	13.54
Efficiency	1.00	0.98	0.90	0.85

Table 1: Analysis done on Thu Dec 29 10:43:04 PM CET 2022, paco1208

Gráfico de dependencias creado con el código de Gauss modificado:



Mismo gráfico una zona con zoom: