Práctica 5

Noel Carmona García

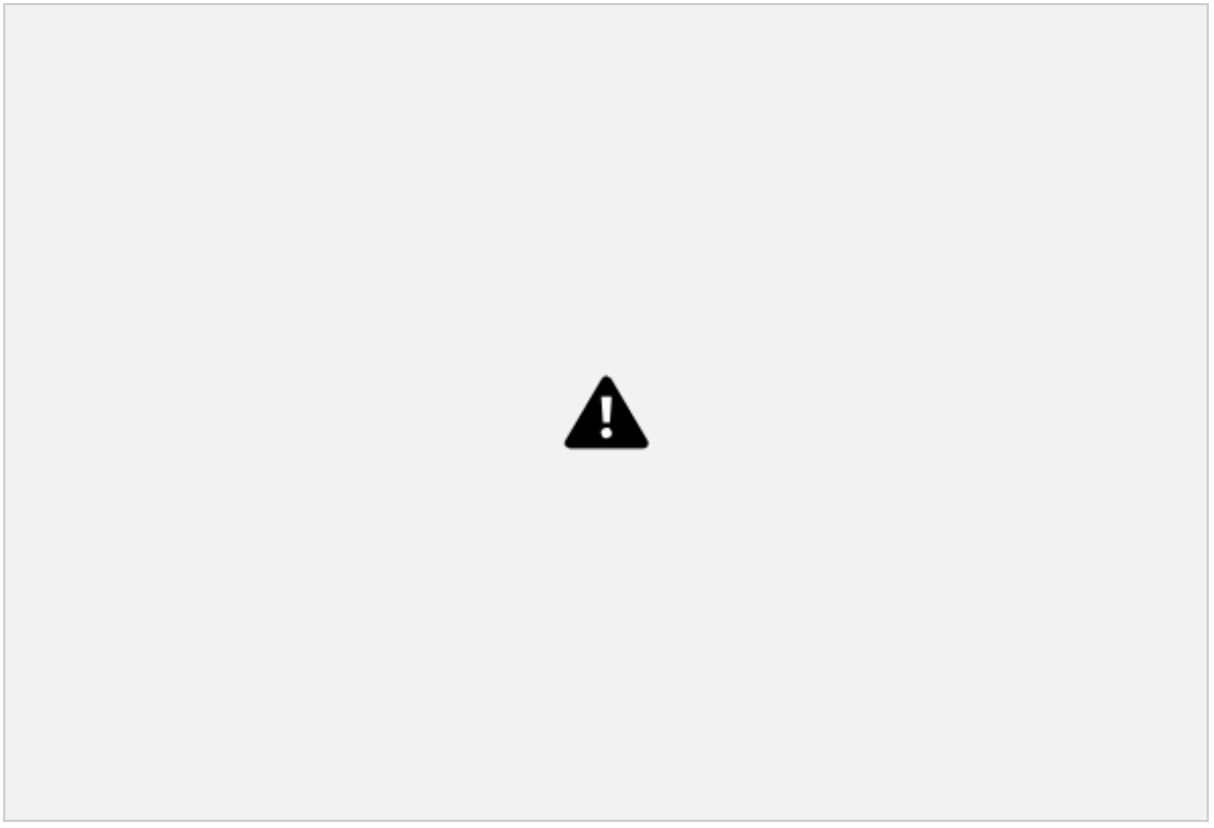
Analysis with tareador

1. Include the relevant parts of the modified solver-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear in the two solvers: Jacobi and Gauss-Seidel. How will you protect them in the parallel OpenMP code?

Jacobi:

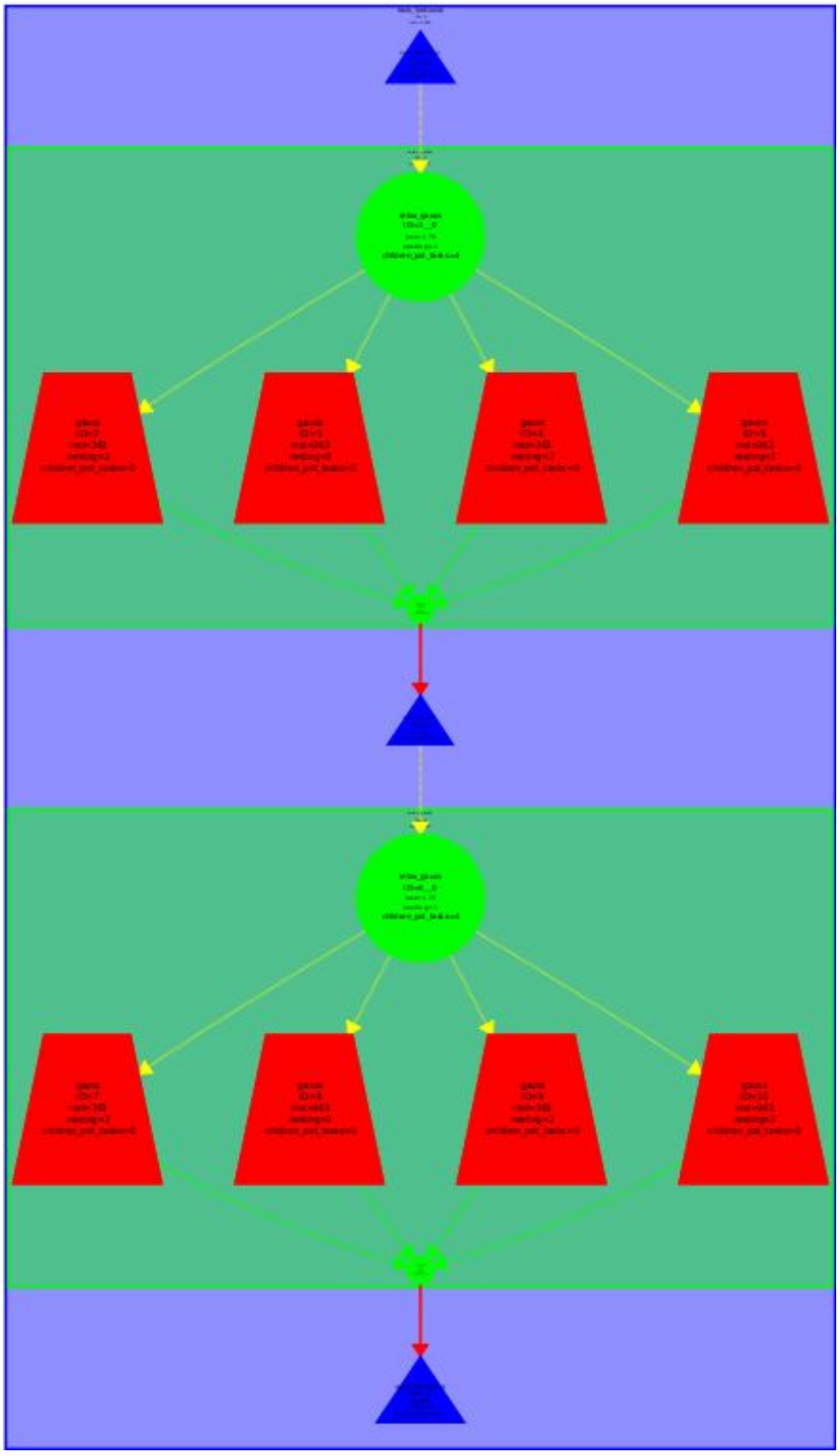
Existe una dependencia en la variable sum, por lo tanto la hemos deshabilitado. En openMP utilizaremos un reduction en la variable sum.

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

    int howmany=8;
    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++) {
        tareador_start_task("jacobi");
        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];
             tareador_disable_object(&sum);
             sum += diff * diff;
             tareador_enable_object(&sum);
        }
        tareador_end_task("jacobi");
      }
    }

    return sum;
}
```



Gauss-Seidel:

Tenemos la dependencia de la variable sum, el resultado se actualiza en la propia matriz, por lo tanto vamos a tener dependencia con la fila superior, este código es menos paralelizable que Jacobi.



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

    int howmany=8;
    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++) {
        tareador_start_task("jacobi");
        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];
             tareador_disable_object(&sum);
             sum += diff * diff;
             tareador_enable_object(&sum);
         }
         tareador_end_task("jacobi");
      }
    }

    return sum;
}
```

OpenMP parallelization and execution analysis: Jacovi

1. Describe the data decomposition strategy that is applied to solve the problem, including a picture with the part of the data structure that is assigned to each processor.

Dividimos las iteraciones del bucle exterior entre el número de threads, cada thread hace todas las iteraciones que tiene asignadas incluido el bucle interior a excepción de la primera y última iteración del bucle.

2. Include the relevant portions of the parallel code that you implemented to solve the heat equation using the Jacobi solver, commenting whatever necessary. Including captures of Paraver windows to justify your explanations and the differences observed in the execution.

Código:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for collapse(2)
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

```
/*
 * Blocked Jacobi solver: one iteration step
 */
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;
}
```

Si paralelizamos la función copy_mat el tiempo de ejecución bajará bastante. En jacobi tenemos una dependencia en la variable sum así que hemos utilizado reduction con la variable y además hemos privatizado la variable diff.

Diferencias en la ejecución:

```
paco11@boada-1:~/lab5$ cat 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: 2.037
Flops and Flops per second: (11.182 GFlop => 5488.06 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
paco11@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: 5.325
Flops and Flops per second: (11.182 GFlop => 2099.97 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
paco11@boada-1:~/lab5$
```

Como podemos ver en la imagen la ejecución secuencial nos toma 5.325, mientras que en la ejecución con el código paralelizado nos menos de la mitad, 2.037.

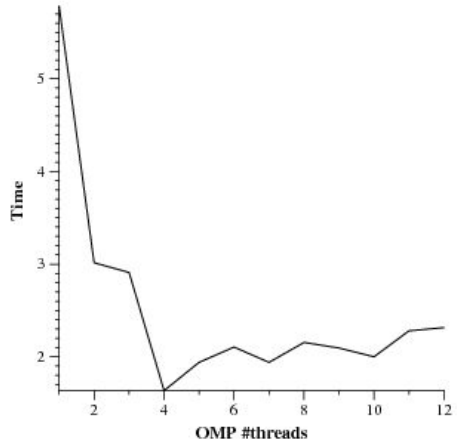
Paraver:

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,847,439,136 ns	-	814,259,515 ns	14,343,706 ns	2,436 ns
THREAD 1.1.2	1,956,185,935 ns	298,761,128 ns	-	5,066,571 ns	-
THREAD 1.1.3	1,703,373,208 ns	298,695,700 ns	-	4,549,583 ns	-
THREAD 1.1.4	1,804,871,554 ns	298,760,743 ns	-	4,190,054 ns	-
THREAD 1.1.5	614,332,994 ns	298,790,019 ns	-	4,315,129 ns	-
THREAD 1.1.6	481,678,701 ns	298,758,545 ns	-	3,746,697 ns	-
THREAD 1.1.7	427,141,548 ns	298,706,868 ns	-	3,632,180 ns	-
THREAD 1.1.8	746,436,703 ns	298,816,035 ns	-	3,713,303 ns	-
Total	9,581,459,779 ns	2,091,289,038 ns	814,259,515 ns	43,557,223 ns	2,436 ns
Average	1,197,682,472.38 ns	298,755,576.86 ns	814,259,515 ns	5,444,652.88 ns	2,436 ns
Maximum	1,956,185,935 ns	298,816,035 ns	814,259,515 ns	14,343,706 ns	2,436 ns
Minimum	427,141,548 ns	298,695,700 ns	814,259,515 ns	3,632,180 ns	2,436 ns
StDev	639,553,262.12 ns	39,398.25 ns	0 ns	3,393,979.40 ns	0 ns
Avg/Max	0.61	1.00	1	0.38	1

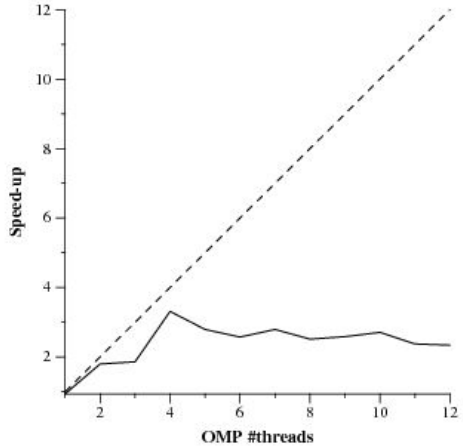
En la tabla del paraver podemos observar que los cuatro primeros procesadores trabajan más que los otros cuatro, esto se debe a que la granulación es demasiado alta y es difícil hacer una mejor repartición de tareas. El bucle principal tiene 4 iteraciones y las reparte entre los 4 procesadores.

Podemos mejorar el rendimiento si aumentamos el valor de la variable howmany, el trabajo se repartirá mejor entre los 8 procesadores.

3. Include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed.



Average elapsed execution time



Speed-up wrt sequential time

En las gráficas podemos observar como en el caso de 4 procesadores obtenemos un mejor tiempo de ejecución y el speed-up más alto. Este grafo demuestra lo comentado en el apartado anterior, si queremos un mejor rendimiento vamos a tener que utilizar más threads y aumentar el valor de la variable howmany.

OpenMP parallelization and execution analysis: Gauss-Seidel

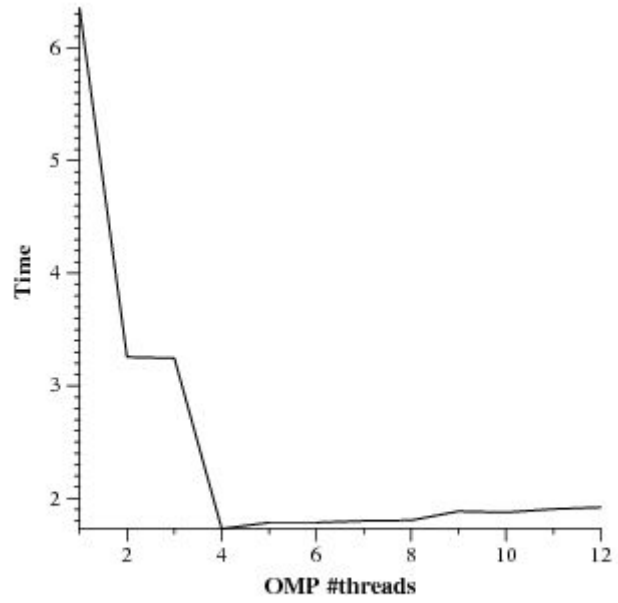
1. Include the relevant portions of the parallel code that implements the Gauss-Seidel solver, commenting how you implemented the synchronization between threads.

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

    int howmany=4;
    #pragma omp parallel for private(diff, unew) 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++) {
            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;
        }
      }
    }

    return sum;
}
```

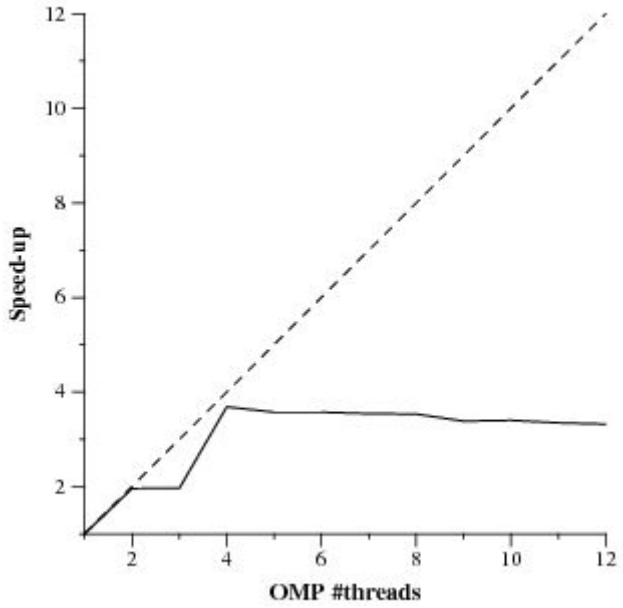
A la hora de paralelizar el código no cambia mucho respecto a la versión Jacobi, en este caso también vamos a tener que privatizar el el valor de la variable unew

2. Include the speed–up (strong scalability) plot that has been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

howmany=4



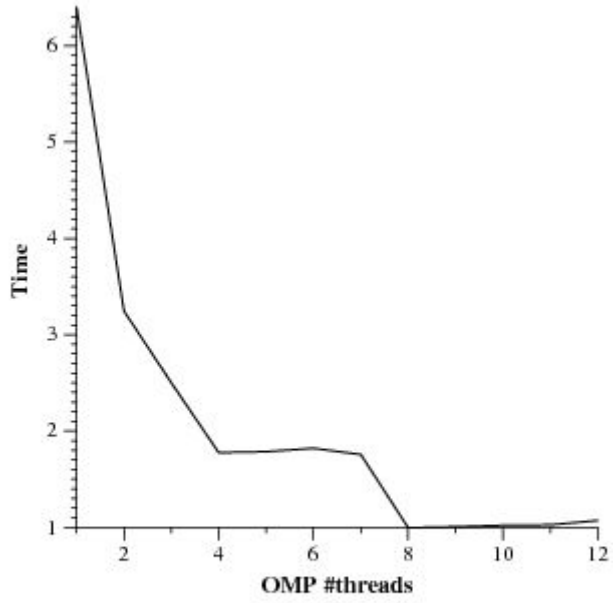
Average elapsed execution time



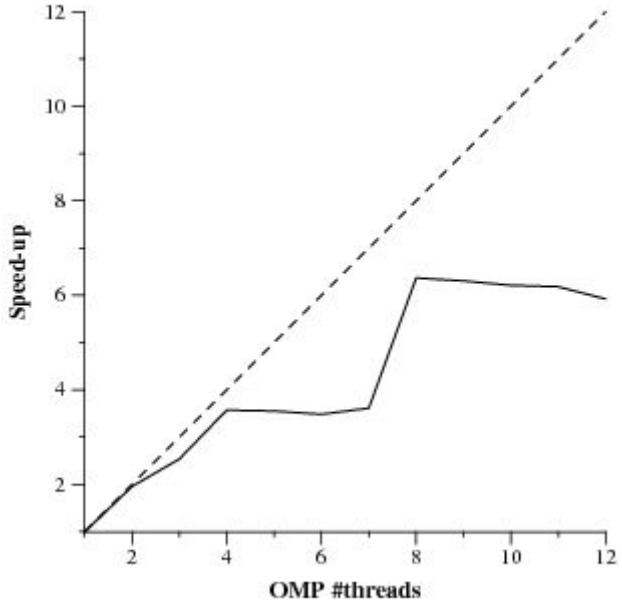
Speed-up wrt sequential time

Tenemos una repartición de datos basada en el número asignado a howmany, si tenemos más threads estos no tendrán tareas, esto lo podemos ver en la anterior gráfica, en la cual howmany equivale a 4 y en la mostrada a continuación, la cual tiene howmany=8

howmany=8



Average elapsed execution time



Speed-up wrt sequential time

Con paraver podemos ver mejor como se hace la repartición de tareas:

Paraver (howmany=4):

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	2,045,008,906 ns	-	152,021,826 ns	6,893,319 ns	2,455 ns
THREAD 1.1.2	1,691,860,168 ns	295,752,615 ns	-	1,600,555 ns	-
THREAD 1.1.3	1,745,348,991 ns	295,648,951 ns	-	1,597,777 ns	-
THREAD 1.1.4	1,689,555,758 ns	295,655,521 ns	-	1,483,904 ns	-
THREAD 1.1.5	31,397,261 ns	295,648,933 ns	-	1,524,306 ns	-
THREAD 1.1.6	31,664,662 ns	295,648,916 ns	-	1,481,216 ns	-
THREAD 1.1.7	30,659,861 ns	295,648,816 ns	-	1,424,975 ns	-
THREAD 1.1.8	29,513,226 ns	295,648,906 ns	-	1,433,885 ns	-
Total	7,295,008,833 ns	2,069,652,658 ns	152,021,826 ns	17,439,937 ns	2,455 ns
Average	911,876,104.12 ns	295,664,665.43 ns	152,021,826 ns	2,179,992.12 ns	2,455 ns
Maximum	2,045,008,906 ns	295,752,615 ns	152,021,826 ns	6,893,319 ns	2,455 ns
Minimum	29,513,226 ns	295,648,816 ns	152,021,826 ns	1,424,975 ns	2,455 ns
StDev	887,197,080.39 ns	35,977.79 ns	0 ns	1,782,542.07 ns	0 ns
Avg/Max	0.45	1.00	1	0.32	1

Paraver (howmany=8):

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,209,043,286 ns	-	165,985,131 ns	5,807,334 ns	2,250 ns
THREAD 1.1.2	917,096,161 ns	276,978,074 ns	-	1,489,937 ns	-
THREAD 1.1.3	924,931,572 ns	276,965,111 ns	-	1,925,467 ns	-
THREAD 1.1.4	906,266,267 ns	276,961,906 ns	-	1,354,471 ns	-
THREAD 1.1.5	901,943,053 ns	276,965,131 ns	-	1,436,599 ns	-
THREAD 1.1.6	910,654,123 ns	276,965,084 ns	-	1,406,683 ns	-
THREAD 1.1.7	890,235,285 ns	276,965,049 ns	-	1,441,545 ns	-
THREAD 1.1.8	870,216,989 ns	276,961,934 ns	-	1,404,956 ns	-
Total	7,530,386,736 ns	1,938,762,289 ns	165,985,131 ns	16,266,992 ns	2,250 ns
Average	941,298,342 ns	276,966,041.29 ns	165,985,131 ns	2,033,374 ns	2,250 ns
Maximum	1,209,043,286 ns	276,978,074 ns	165,985,131 ns	5,807,334 ns	2,250 ns
Minimum	870,216,989 ns	276,961,906 ns	165,985,131 ns	1,354,471 ns	2,250 ns
StDev	102,418,333.02 ns	5,103.94 ns	0 ns	1,436,345.35 ns	0 ns
Avg/Max	0.78	1.00	1	0.35	1

3. Explain how did you obtain the optimum value for the ratio computation/synchronization in the parallelization of this solver for 8 threads.

Para obtener una buena optimización con 8 threads hay que asignarle la variable howmany 8 ya que tenemos que igualar el número de bloques con el número de threads