Fifth Deliverable

Marlen Avila
Ricard Meyerhofer
Julita Corbalán
PAR2104 2015-16Q1

5.1 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?

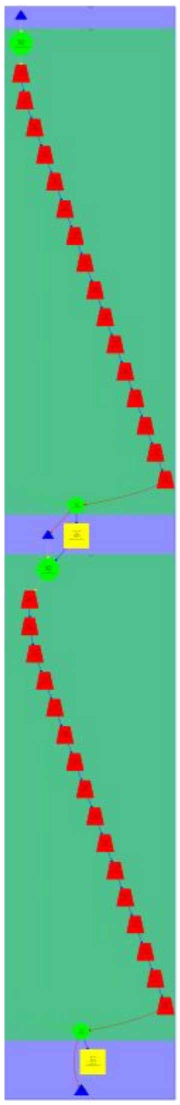
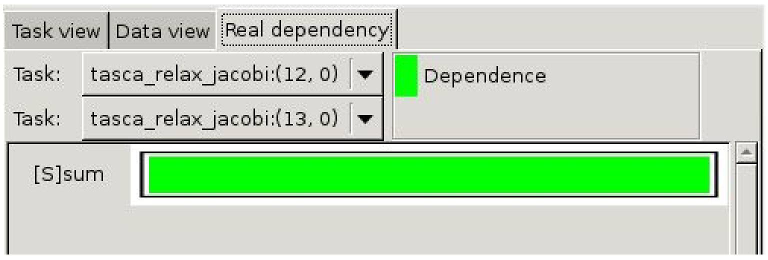
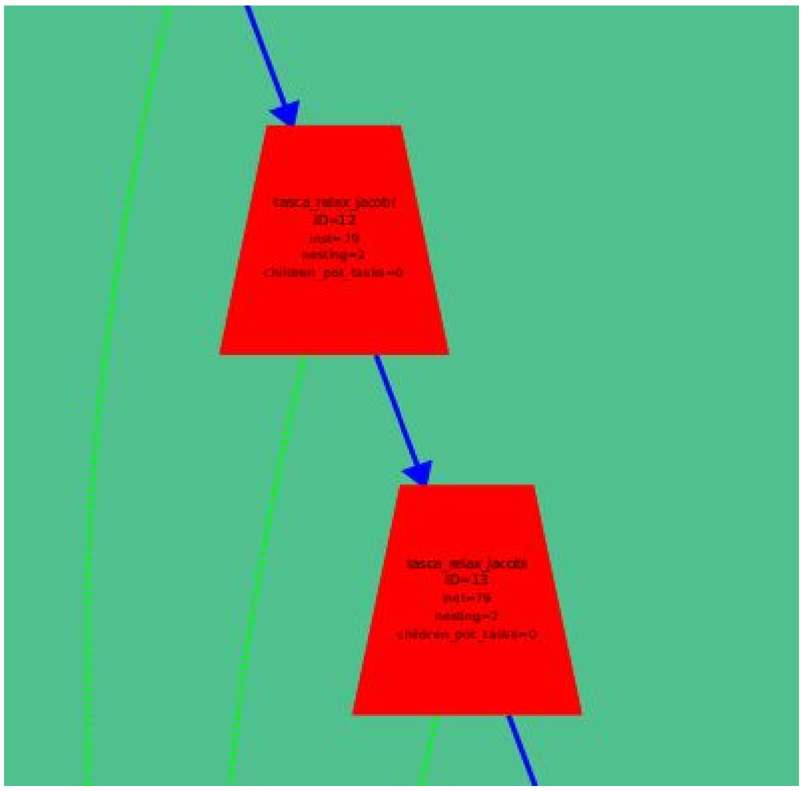
 **Figure 0:** Task Graph generated by Jacobi.	 Task: tasca_relax_jacobi:(12, 0)
	Task: tasca_relax_jacobi:(13, 0)
	🟩 Dependence
	[S]sum 🟩
	Figure 1: Jacobi data dependences.
	 tasca_relax_jacobi
	ID=12
	inst=79
	nesting=2
	children_pot_tasks=0
	tasca_relax_jacobi
	ID=13
	inst=79
	nesting=2
	children_pot_tasks=0 **Figure 2:** Zoom data dependence.

Table 0: Zoom data dependence.

Jacobi instrumentation

```
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("tasca_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];
                 tareador_disable_object(&sum);
                 sum += diff * diff;
                 tareador_enable_object(&sum);
                tareador_end_task("tasca_relax_jacobi");
              }
           }
        }

        return sum;
}
```

As we can see in the dependence task graph above (Table 1), we have a dependence that prevent us from parallelizing the code. That variable is “sum” (Table 1, Figure 1) and once we’ve disabled it, the task dependence graph we have now is the one shown in Figure 3, where all the tasks now are potentially parallelizable.

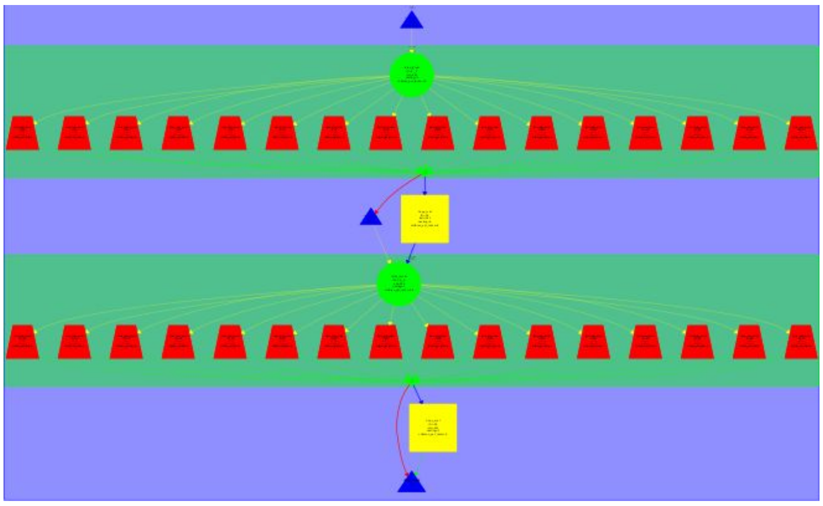


Figure 3: Jacobi instrumented code with the sum variable disabled.

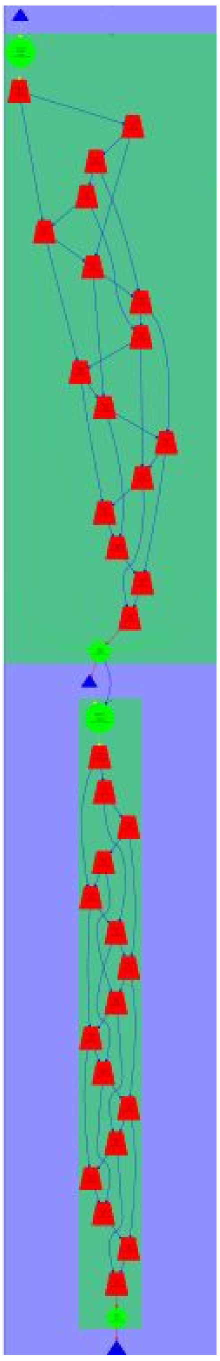
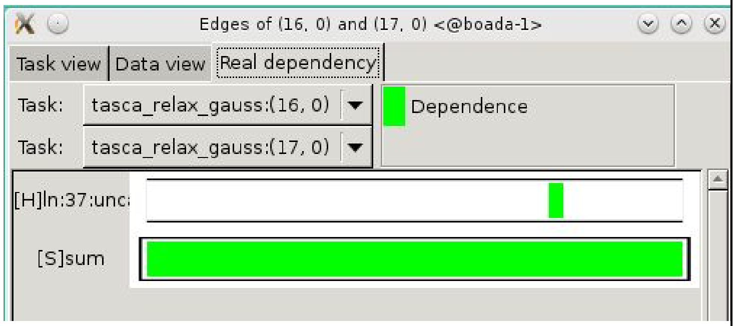


Figure 4: Task Graph generated by Gauss-Seidel.



Edges of (16, 0) and (17, 0) <@boada-1>

Task view | Data view | Real dependency

Task: tasca_relax_gauss:(16, 0)

Task: tasca_relax_gauss:(17, 0)

🟩 Dependence

[H]ln:37:unc

[S]sum

Figure 5: Gauss-Seidel data dependences.

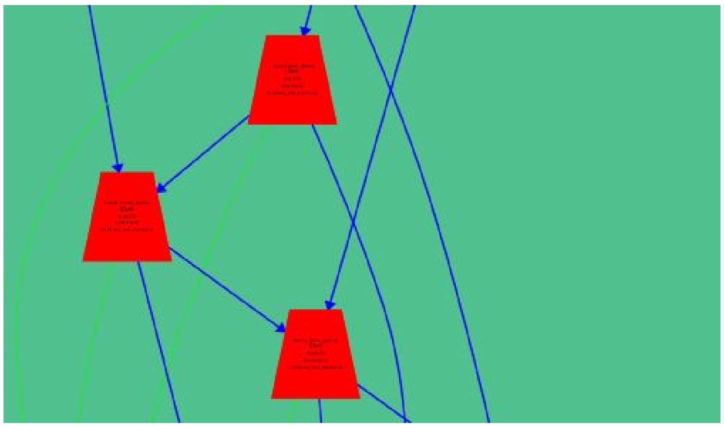


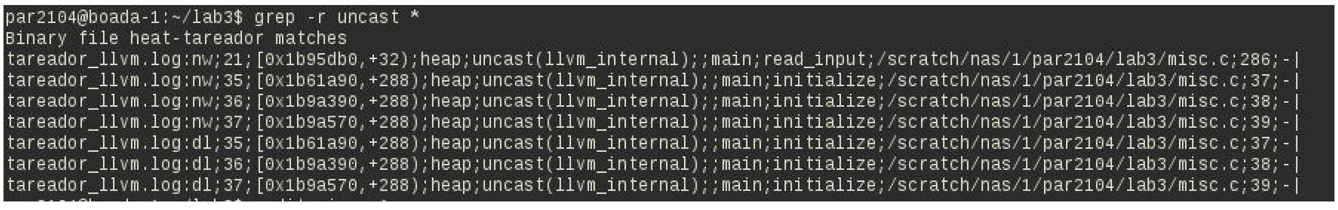
Figure 6: Zoom Gauss-Seidel.

Table 1: Zoom data dependence.

Gauss-Seidel instrumentation

```
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("tasca_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];
              tareador_disable_object(&sum);
              sum += diff * diff;
              u[i*sizey+j]=unew;
              tareador_enable_object(&sum);
              tareador_end_task("tasca_relax_gauss");
           }
         }
       }
       return sum;
}
```

In this case, we have more than one variable dependence (see Table 1). One of the dependences is the variable sum again, so we can disable it as before. On the other hand, the dependence that creates the variable uncast is a fix one, so it’s impossible to disable it and make the code totally parallelizable. The result graph is the one shown in Figure 4, and in the Figure 3 we can see the grep command we’ve used to find the uncast variable.



```
par2104@boada-1:~/lab3$ grep -r uncast *
Binary file heat-tareador matches
tareador_llvm.log:nw;21;[0x1b95db0,+32);heap;uncast(llvm_internal);;main;read_input;/scratch/nas/1/par2104/lab3/misc.c;286;-|
tareador_llvm.log:nw;35;[0x1b61a90,+288);heap;uncast(llvm_internal);;main;initialize;/scratch/nas/1/par2104/lab3/misc.c;37;-|
tareador_llvm.log:nw;36;[0x1b9a390,+288);heap;uncast(llvm_internal);;main;initialize;/scratch/nas/1/par2104/lab3/misc.c;38;-|
tareador_llvm.log:nw;37;[0x1b9a570,+288);heap;uncast(llvm_internal);;main;initialize;/scratch/nas/1/par2104/lab3/misc.c;39;-|
tareador_llvm.log:dl;35;[0x1b61a90,+288);heap;uncast(llvm_internal);;main;initialize;/scratch/nas/1/par2104/lab3/misc.c;37;-|
tareador_llvm.log:dl;36;[0x1b9a390,+288);heap;uncast(llvm_internal);;main;initialize;/scratch/nas/1/par2104/lab3/misc.c;38;-|
tareador_llvm.log:dl;37;[0x1b9a570,+288);heap;uncast(llvm_internal);;main;initialize;/scratch/nas/1/par2104/lab3/misc.c;39;-|
```

Figure 3: Grep command output

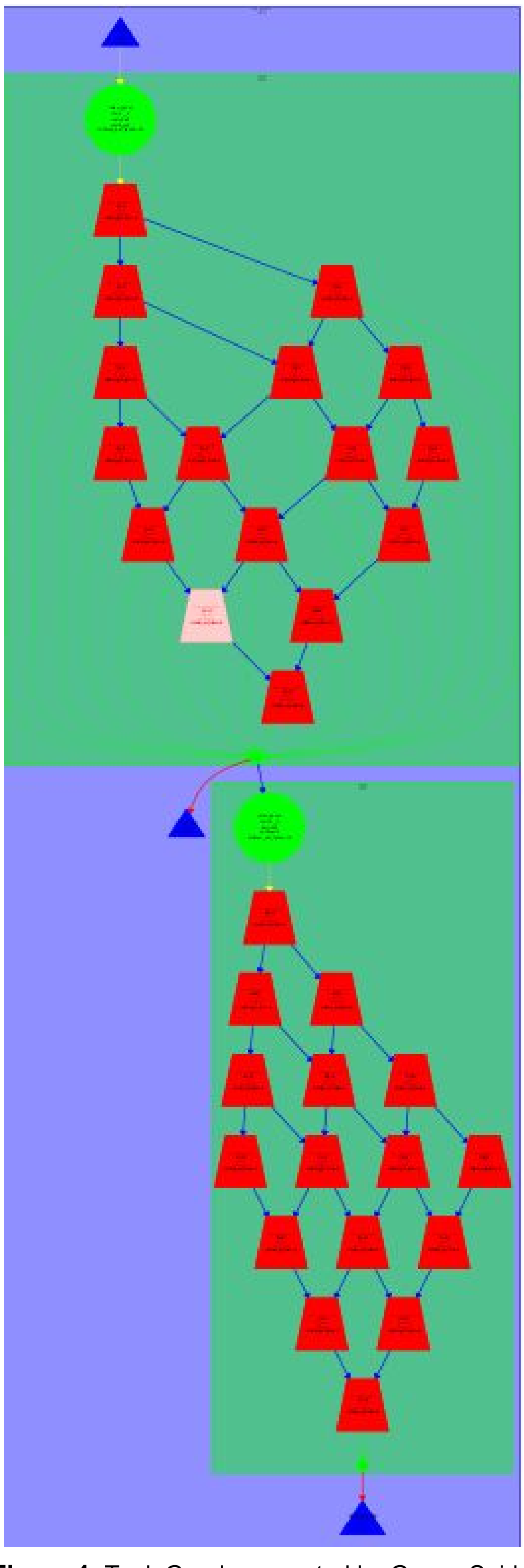


Figure 4: Task Graph generated by Gauss-Seidel

The OpenMP clauses that we'd use would be a **#pragma omp parallel for reduction (+:sum) private (diff)** in both cases (jacobi and gauss). With the **for** clause the loop will be executed by multiples threads at the same time. The **reduction** clause with the variable sum is needed in order to accumulate the partial results of the different threads avoiding data race. The **private** clause with the diff variable is also necessary to avoid data race (this way the variable is exclusive for each thread), since the sum result is calculated with diff.

Copy_mat instrumentation

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
        for (int i=1; i<= sizex-2; i++)
           for (int j=1; j<= sizey-2; j++) {
               tareador_start_task("copy");
               v[ i*sizey+j ] = u[ i*sizey+j ];
               tareador_end_task("copy");
            }
}
```

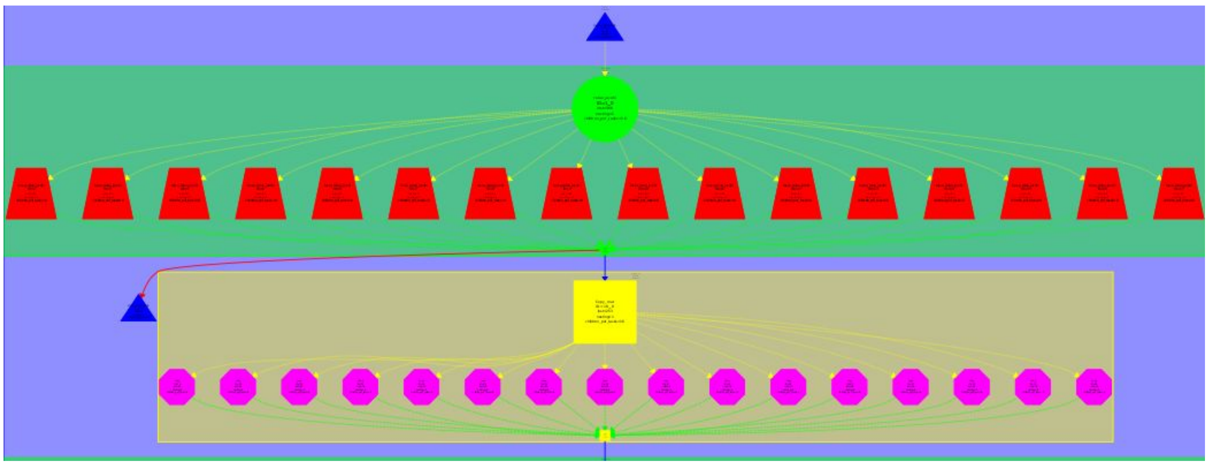
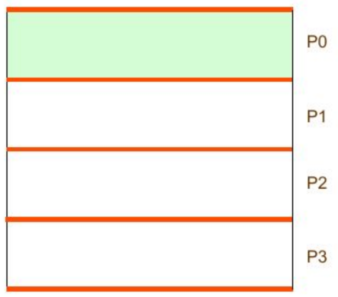


Figure 6: Copy mat potential parallelization

In addition to the instrumentation of the two versions of the code (Jacobi and Gauss) , we've seen that the copy_mat function doesn't have any dependences (see Figure 6 above) so it's totally parallelizable.

5.2 OpenMP parallelization and execution analysis: Jacobi

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.
R/
The data decomposition strategy applied to solve the problem is a Block Data Decomposition. The main concept is about taking advantage of each thread working in its maximum load.



[2,3] -
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.

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.
R/

Jacobi OpenMP parallelization
```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
        double diff, sum=0.0;
        int howmany=4;
        #pragma omp parallel for reduction (+:sum) private (diff)
        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;} //sino quedaba fea la tabla
```

Copy_mat OpenMP parallelization

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];
}

To parallelize the Jacobi function we've added the clause in bold seen in the table above (**Jacobi OpenMP parallelization**). With the for clause the loop is executed by multiple threads at the same time, which is the base of our parallelization. The variable sum calculates the final result, so is necessary to avoid a data race issue, which is why we've added the reduction clause. This way, each thread will calculate its own partial result and then it will be accumulated in the sum variable without overwriting the other thread's results. For the same reason, the private clause is needed because the sum variable uses the diff one to calculate its result, so this way we're avoiding again another data race condition.
In the second table (**Copy_mat OpenMP parallelization**) we've parallelized the copy_mat function too, since we've seen in Figure 6 (question 5.1) that there are no dependences. Again we've used the for clause but this time combined with a collapse that does a better task repartition.
Once the code is parallelized, we proceed to analyze its performance:

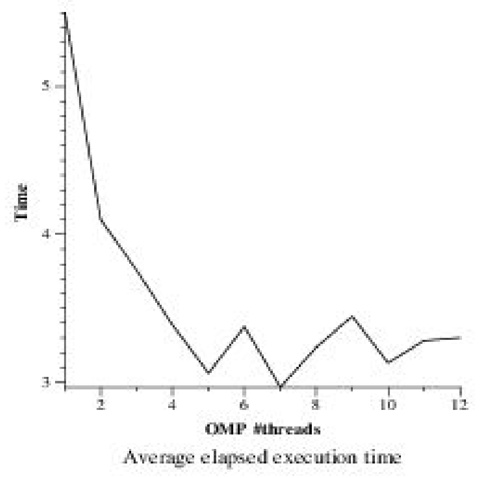
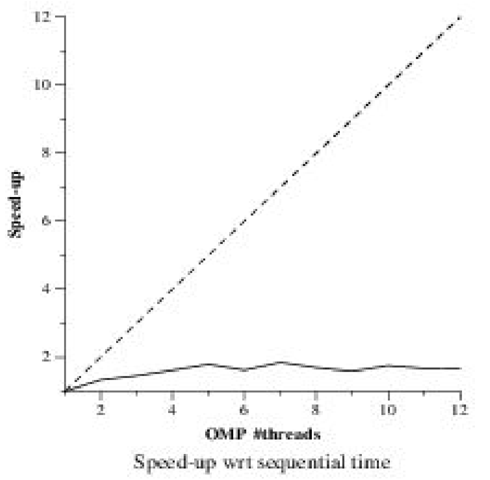
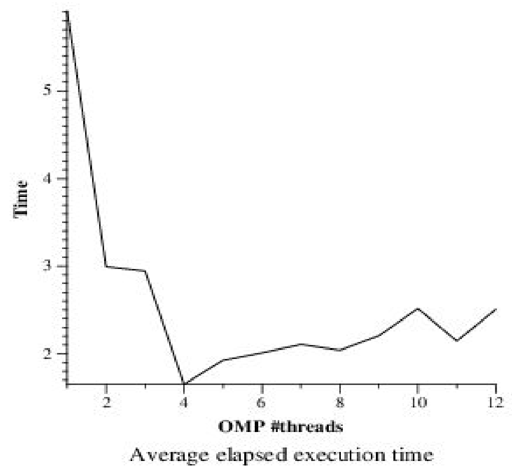
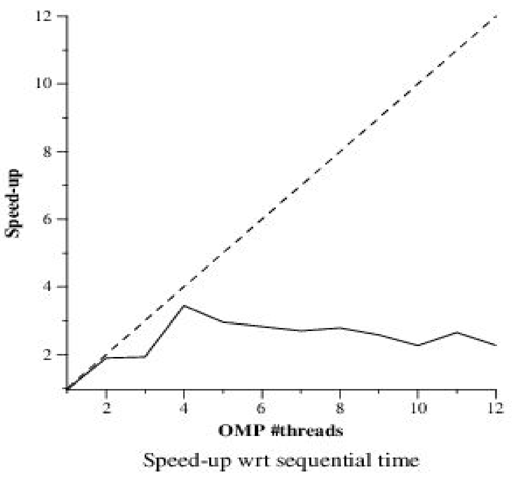
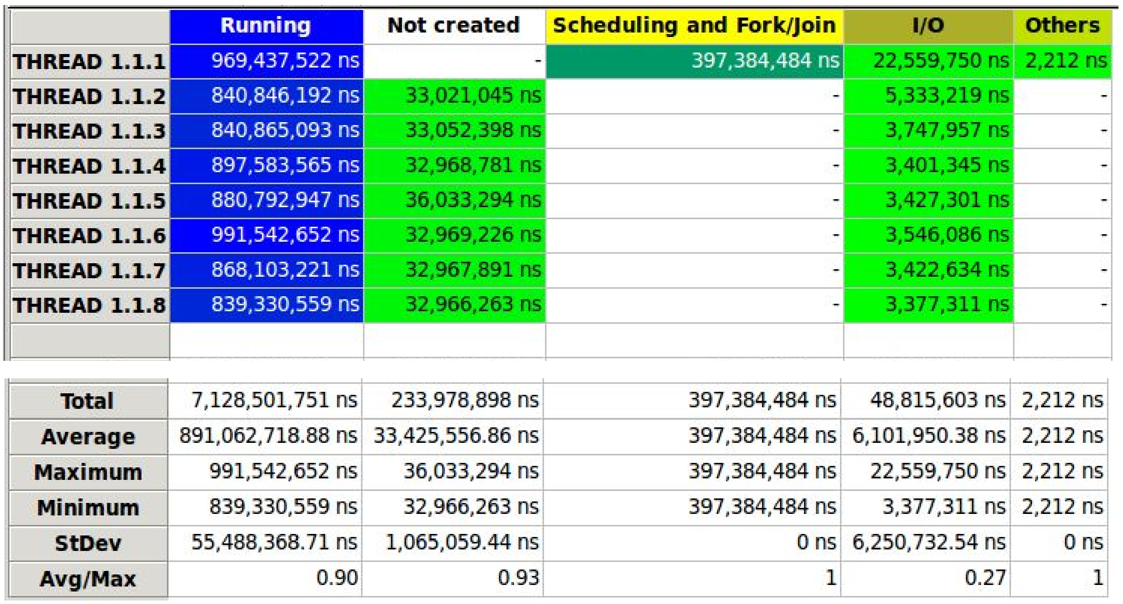
Strong-omp.sh plots of Jacobi parallelization without copy_mat parallelization	**Strong-omp.sh plots of Jacobi parallelization with copy_mat parallelitzation**
 	 

Table 3: Plot generated by submit-strong-omp.sh of the first parallelized version (right) and the same version without the copy_mat parallelization (left).

As we can see in Table 3 right side, the performance is not as good as we thought it would be by parallelizing the first loop. The speed-up is far from being adjust to the ideal curve once we reach 4 threads (see speed-up plot), so we're going to throw a paraver trace in order to see if we can figure out which is the problem.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	969,437,522 ns	-	397,384,484 ns	22,559,750 ns	2,212 ns
THREAD 1.1.2	840,846,192 ns	33,021,045 ns	-	5,333,219 ns	-
THREAD 1.1.3	840,865,093 ns	33,052,398 ns	-	3,747,957 ns	-
THREAD 1.1.4	897,583,565 ns	32,968,781 ns	-	3,401,345 ns	-
THREAD 1.1.5	880,792,947 ns	36,033,294 ns	-	3,427,301 ns	-
THREAD 1.1.6	991,542,652 ns	32,969,226 ns	-	3,546,086 ns	-
THREAD 1.1.7	868,103,221 ns	32,967,891 ns	-	3,422,634 ns	-
THREAD 1.1.8	839,330,559 ns	32,966,263 ns	-	3,377,311 ns	-
Total	7,128,501,751 ns	233,978,898 ns	397,384,484 ns	48,815,603 ns	2,212 ns
Average	891,062,718.88 ns	33,425,556.86 ns	397,384,484 ns	6,101,950.38 ns	2,212 ns
Maximum	991,542,652 ns	36,033,294 ns	397,384,484 ns	22,559,750 ns	2,212 ns
Minimum	839,330,559 ns	32,966,263 ns	397,384,484 ns	3,377,311 ns	2,212 ns
StDev	55,488,368.71 ns	1,065,059.44 ns	0 ns	6,250,732.54 ns	0 ns
Avg/Max	0.90	0.93	1	0.27	1

Figure 7: Paraver trace of Jacobi parallelization

It can be seen in the paraver trace that there is a work unbalance between threads (for example, the first 4 threads spend more time working than the other four ones). We'll try to fix this problem by reducing the granularity of the tasks executed by each thread so this way we can increase our load balance and as result we can increase our perfomance. **How can this be done?**

Optimization of code

```
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  reduction (+:sum) private (diff)
    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;}
```

The reason why we use the **omp_get_max_threads()** call is because it returns the maximum number of threads that are available to work in a parallel region, so when we're distributing in blocks all the work we will consider the maximum number of threads available instead of a fixed number. This way, all the workloads of each thread will be more balanced.
Now we are going to generate the speed-up and time plots with the submit-strong.sh script and analyze the paraver traces:

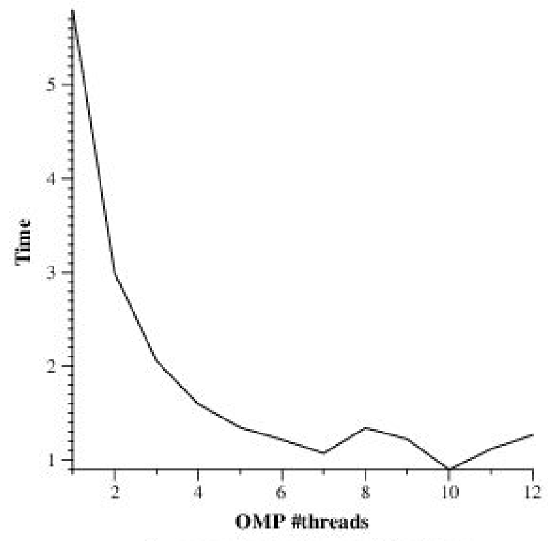
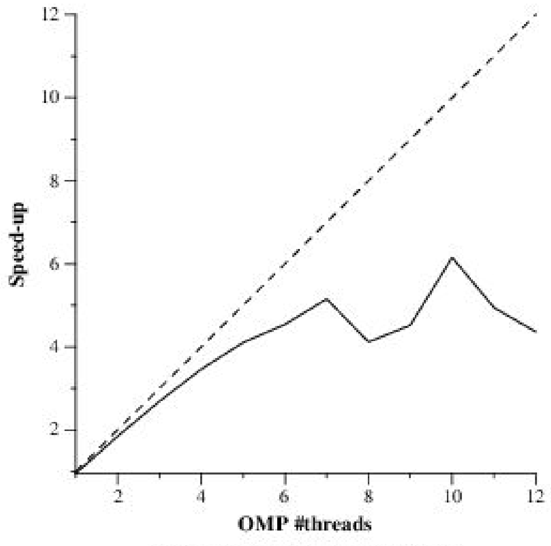
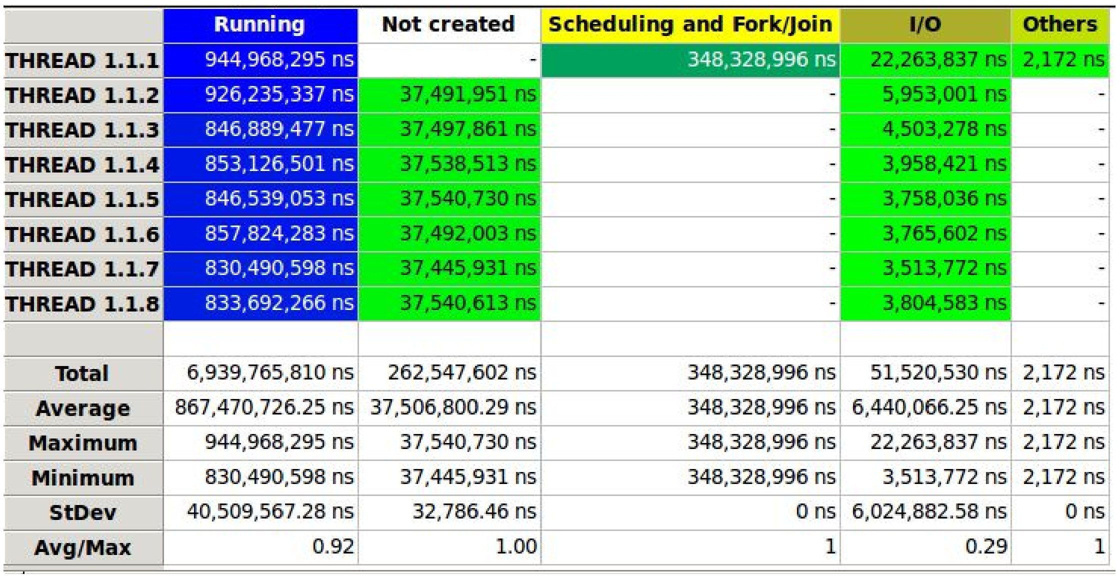
Jacobi OpenMP parallelization improvement
 Average elapsed execution time    Speed-up wrt sequential time

Table 4: Plot with the improvement



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	944,968,295 ns	-	348,328,996 ns	22,263,837 ns	2,172 ns
THREAD 1.1.2	926,235,337 ns	37,491,951 ns	-	5,953,001 ns	-
THREAD 1.1.3	846,889,477 ns	37,497,861 ns	-	4,503,278 ns	-
THREAD 1.1.4	853,126,501 ns	37,538,513 ns	-	3,958,421 ns	-
THREAD 1.1.5	846,539,053 ns	37,540,730 ns	-	3,758,036 ns	-
THREAD 1.1.6	857,824,283 ns	37,492,003 ns	-	3,765,602 ns	-
THREAD 1.1.7	830,490,598 ns	37,445,931 ns	-	3,513,772 ns	-
THREAD 1.1.8	833,692,266 ns	37,540,613 ns	-	3,804,583 ns	-
Total	6,939,765,810 ns	262,547,602 ns	348,328,996 ns	51,520,530 ns	2,172 ns
Average	867,470,726.25 ns	37,506,800.29 ns	348,328,996 ns	6,440,066.25 ns	2,172 ns
Maximum	944,968,295 ns	37,540,730 ns	348,328,996 ns	22,263,837 ns	2,172 ns
Minimum	830,490,598 ns	37,445,931 ns	348,328,996 ns	3,513,772 ns	2,172 ns
StDev	40,509,567.28 ns	32,786.46 ns	0 ns	6,024,882.58 ns	0 ns
Avg/Max	0.92	1.00	1	0.29	1

Figure 8: Paraver trace of Jacobi parallelization with the howmany modified generated by the submit-omp-i.sh script.

As we can see our speed-up plot is now much better than the one seen in table 3 right side (especially in the results for more than 4 threads). In addition if we compare the two paraver traces, we can see that the time execution and work balance are way better (just like we expected) thanks to our block distribution.

5.3 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.

R/

Gauss-Siedel OpenMP parallelization

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
        double unew, diff, sum=0.0;
        int howmany=omp_get_max_threads();
         int howmanyAux = howmany; //used for the optimum value search
        int finished[howmany];
        #pragma omp parallel for
        for (int i = 0; i<howmany; i++) finished[i] = 0;
        #pragma omp parallel for reduction(+:sum) private(diff, unew)
        for (int blockid = 0; blockid < howmany; ++blockid) {
        int i_start = lowerb(blockid, howmany, sizex);
        int i_end = upperb(blockid, howmany, sizex);
        for (int z = 0; z < howmanyAux; z++) {
              int j_start = lowerb(z, howmanyAux, sizey);
              int j_end = upperb(z,howmanyAux, sizey);
              if (blockid > 0) {
                   while (finished[blockid-1] <= z) {
                     #pragma omp flush
                   }
              }
             for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                  for (int j = max(1, j_start); j<= min(j_end, 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;
                    }
            }
               finished[blockid]++;
                #pragma omp flush
             }
        }
        return sum;
}
```

First of all, we’ve modified the #pragma by adding the unew variable (which represents the auxiliary matrix) in the private clause, since it is used to calculate the final result and we need to avoid a data race condition once again.
Secondly, we’ll proceed to describe how the code works: the main idea is to divide the rows each thread has into blocks in order to improve the performance and the time execution. The value that will determine the division of blocks is the howmanyAux one, that in this case will be fixed with the howmany value, but we’ll discuss how affects this variable to the result in the next point 5.3.3.
We have two type of dependencies: the left and the top ones. In this case, the best way to proceed is with the Wave-front method.
We don’t have to worry about the left dependency because each block is executed sequentally (b0, then b1, then b2..), so every time we’ll go for the bi+1 block, we’ll already have the result of the bi one.
The top dependency is a little more difficult to treat, so we’ve created a vector (finished[howmany] initialized to 0) that controls the blocks that are finished by marking each position with the last block that has been done by each thread (for example, finished[id] will indicate the last block that has been done by the thread indicated by “id”). This way the “depending” threads will be active-waiting until their dependencies are calculated.
Finally we use #pragma omp flush to maintain the coherency of our memory since the changes will be seen by all the threads.

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.
R/

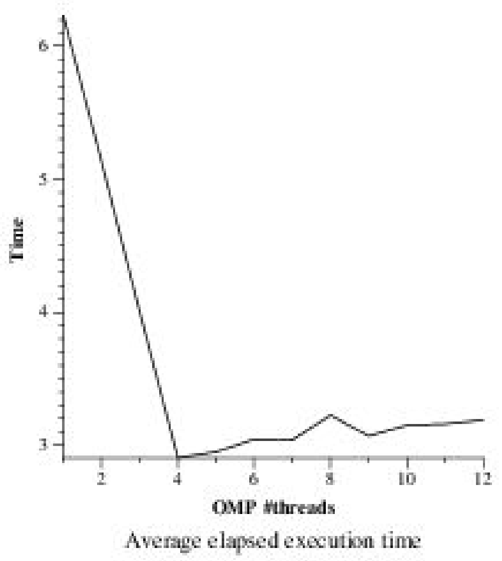
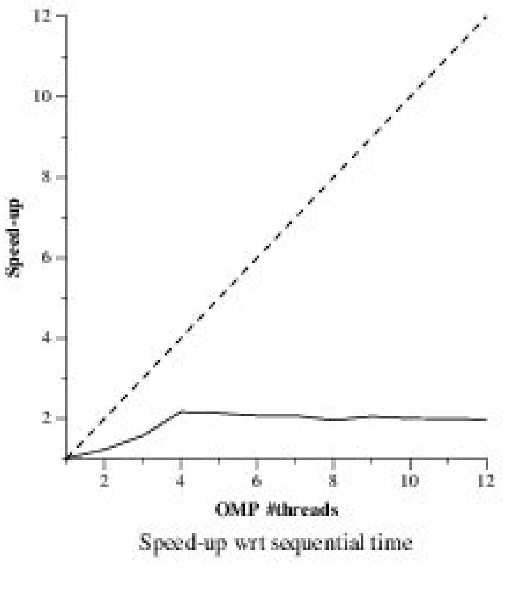
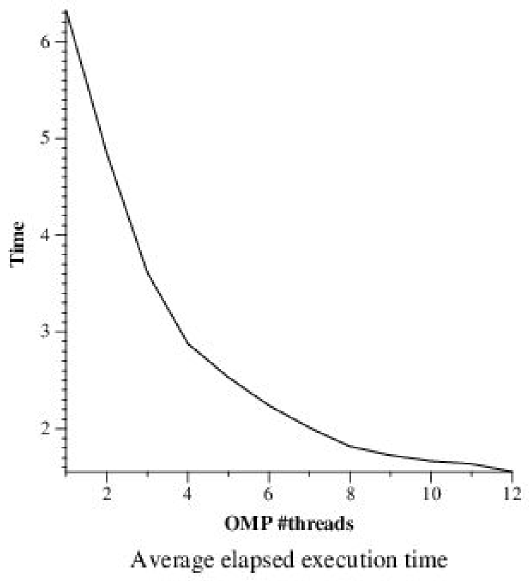
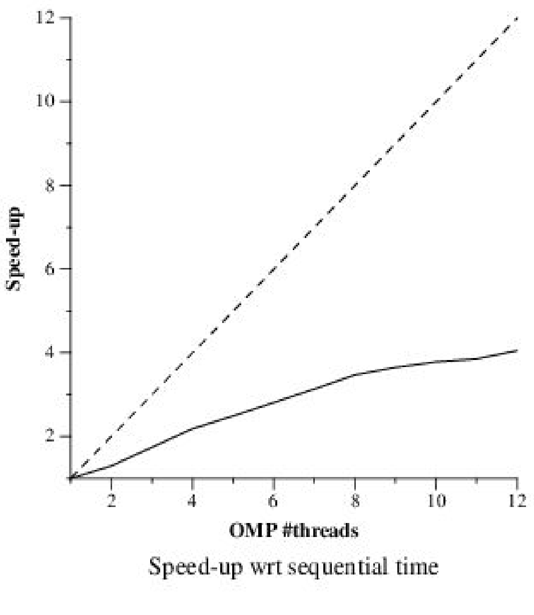
Strong-omp.sh plots of Gauss-Siedel parallelitzation without block distribution	Strong-omp.sh plots of Gauss-Siedel parallelitzation
 	 

Table 5: Plot with the comparation between Gauss-Siedel with and without block distribution.

The reason why in this algorithm the performance is not as good as we expected (is worse than the Jacobi parallelization[Table 4]) is the waiting time that the last thread has due to the dependences mentioned before.

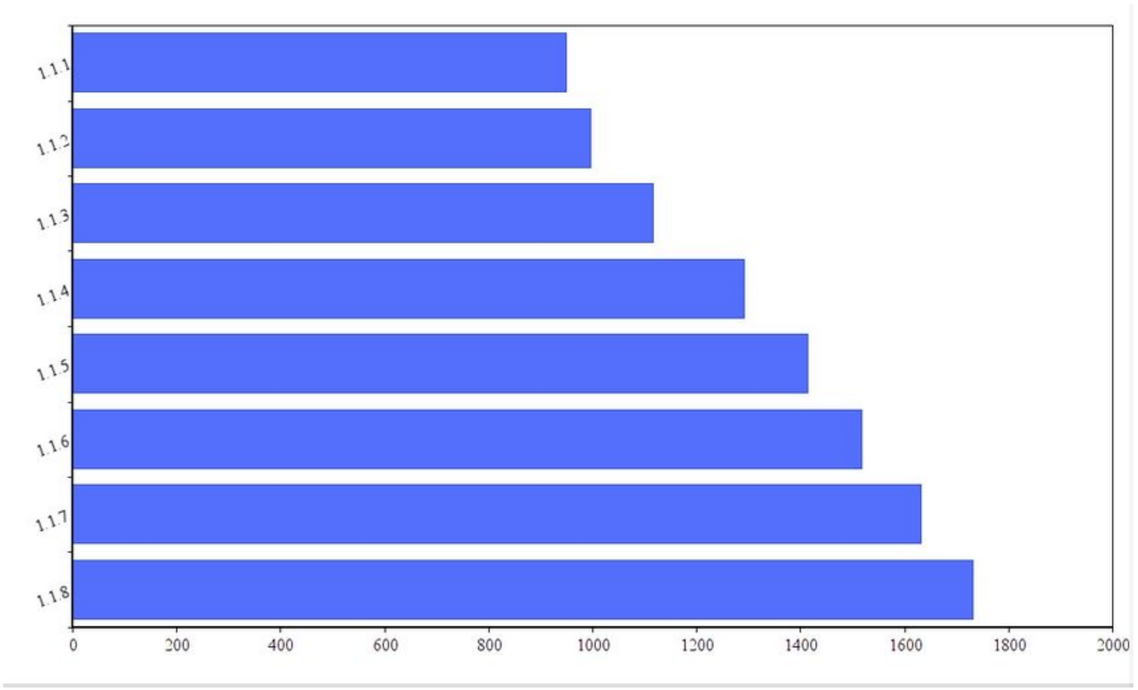


Figure 9: Graphic generated with the column values that appear in figure 11, Y-axis is the threads and X-axis represents the ns divided by 10^6 just to simplificate the scale

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	949,511,881 ns	-	1,067,166,710 ns	16,885,029 ns	2,225 ns
THREAD 1.1.2	996,832,551 ns	33,680,443 ns	-	3,548,374 ns	-
THREAD 1.1.3	1,166,262,198 ns	33,681,330 ns	-	3,875,752 ns	-
THREAD 1.1.4	1,292,896,627 ns	33,615,215 ns	-	2,968,421 ns	-
THREAD 1.1.5	1,413,686,821 ns	33,712,723 ns	-	3,174,094 ns	-
THREAD 1.1.6	1,518,508,819 ns	33,670,515 ns	-	3,134,361 ns	-
THREAD 1.1.7	1,632,990,798 ns	33,709,216 ns	-	3,119,424 ns	-
THREAD 1.1.8	1,732,788,963 ns	33,680,403 ns	-	2,877,758 ns	-
Total	10,703,478,658 ns	235,749,845 ns	1,067,166,710 ns	39,583,213 ns	2,225 ns
Average	1,337,934,832.25 ns	33,678,549.29 ns	1,067,166,710 ns	4,947,901.62 ns	2,225 ns
Maximum	1,732,788,963 ns	33,712,723 ns	1,067,166,710 ns	16,885,029 ns	2,225 ns
Minimum	949,511,881 ns	33,615,215 ns	1,067,166,710 ns	2,877,758 ns	2,225 ns
StDev	269,266,648.65 ns	29,758.14 ns	0 ns	4,521,969.76 ns	0 ns
Avg/Max	0.77	1.00	1	0.29	1

Figure 10: Paraver trace of Gauss-Seidel with howmanyAux = howmany

To improve the performance we will have to look for the best number of blocks that will provide us the best balance between cost of synchronizations and active waiting time (see 5.3 section).

In comparison with the Jacobi parallelization, we see that the Gauss-Seidel is more difficult to parallelize because the Jacobi algorithm does not have a dependency between elements using an auxiliary matrix.

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

We have added to the Gauss-Jacobi code an integer (howmanyAux) that is used to define the number of blocks in which the problem will be divided.
Is important to see that this number will affect the performance of the code depending of its value, as we've said before. If the value of blocks is too large, each of the blocks will have few columns, creating a lot of more needed synchronizations. On its contrary, if the number of blocks is too small, the active-waiting time will be incremented.
Modifying the howmanyAux value, we can check which is the optimum value that in our case is **32**. The values have been obtained by executing the **submit-omp-i.sh** script and look for the result with **cat heat-omp_8.times.txt**.

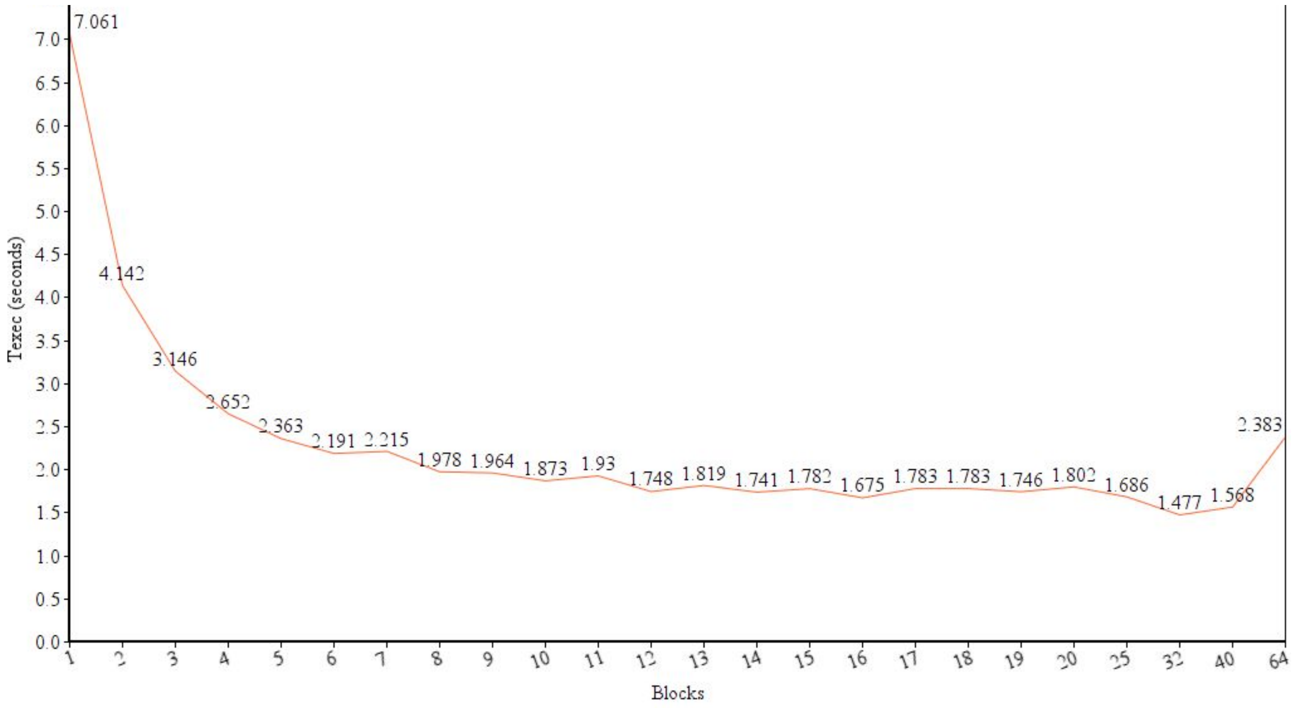
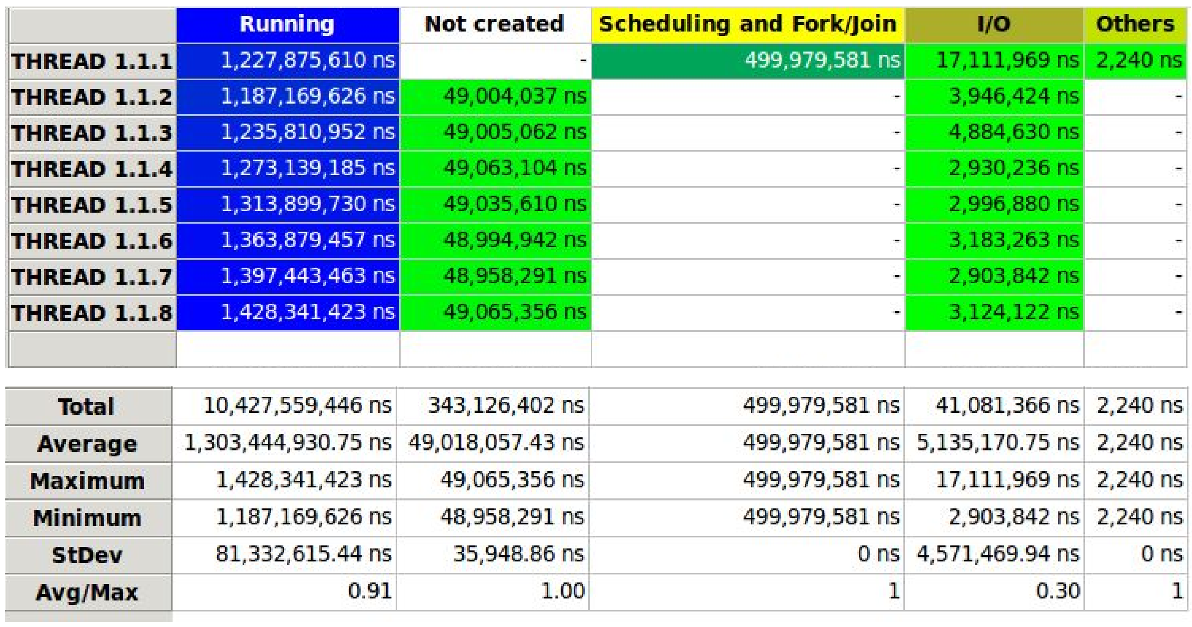


Figure 11: Graphic generated with the values of the table included in the annex (last page).

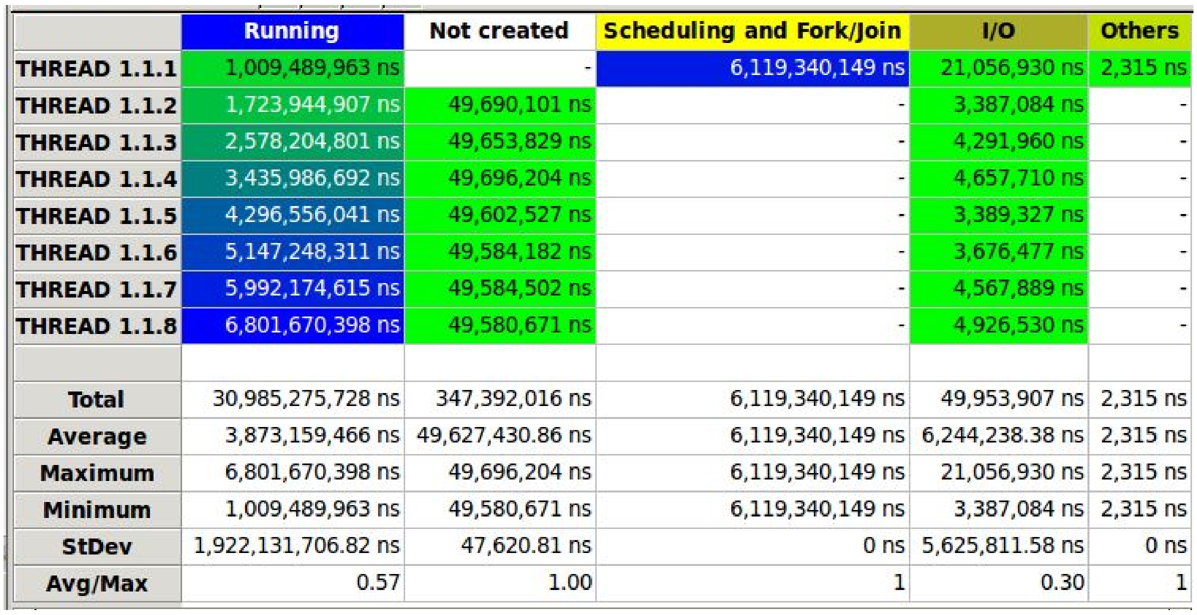
The following paraver trace show us the performance in the best case (howmanyAux = 32), that as we can see it has a better work balance, less scheduling and fork join than the version with howmanyAux = howmany (see Figure 10 above).



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,227,875,610 ns	-	499,979,581 ns	17,111,969 ns	2,240 ns
THREAD 1.1.2	1,187,169,626 ns	49,004,037 ns	-	3,946,424 ns	-
THREAD 1.1.3	1,235,810,952 ns	49,005,062 ns	-	4,884,630 ns	-
THREAD 1.1.4	1,273,139,185 ns	49,063,104 ns	-	2,930,236 ns	-
THREAD 1.1.5	1,313,899,730 ns	49,035,610 ns	-	2,996,880 ns	-
THREAD 1.1.6	1,363,879,457 ns	48,994,942 ns	-	3,183,263 ns	-
THREAD 1.1.7	1,397,443,463 ns	48,958,291 ns	-	2,903,842 ns	-
THREAD 1.1.8	1,428,341,423 ns	49,065,356 ns	-	3,124,122 ns	-
Total	10,427,559,446 ns	343,126,402 ns	499,979,581 ns	41,081,366 ns	2,240 ns
Average	1,303,444,930.75 ns	49,018,057.43 ns	499,979,581 ns	5,135,170.75 ns	2,240 ns
Maximum	1,428,341,423 ns	49,065,356 ns	499,979,581 ns	17,111,969 ns	2,240 ns
Minimum	1,187,169,626 ns	48,958,291 ns	499,979,581 ns	2,903,842 ns	2,240 ns
StDev	81,332,615.44 ns	35,948.86 ns	0 ns	4,571,469.94 ns	0 ns
Avg/Max	0.91	1.00	1	0.30	1

Figure 12: Paraver trace of Gauss-Seidel with howmanyAux = 32 (optimal value)

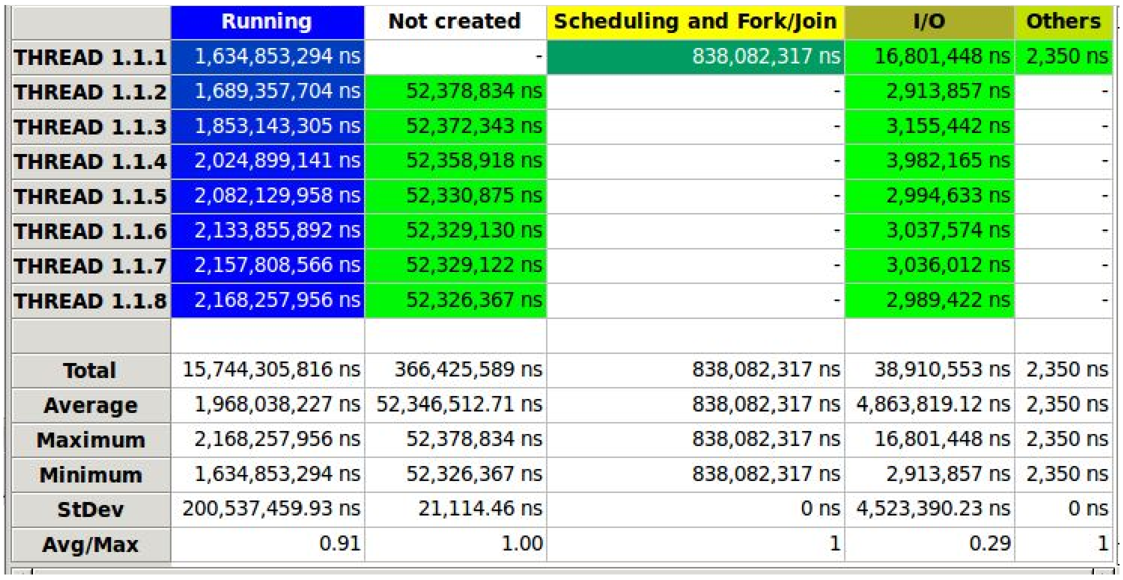
As we can see in the following paraver trace, when the number is really small (1 in this case) the active-waiting time will be incremented which will create a work-unbalance between threads.
We can see that in the worst case 1.1.8 thread spends nearly 7 times more than the 1.1.1 thread.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,009,489,963 ns	-	6,119,340,149 ns	21,056,930 ns	2,315 ns
THREAD 1.1.2	1,723,944,907 ns	49,690,101 ns	-	3,387,084 ns	-
THREAD 1.1.3	2,578,204,801 ns	49,653,829 ns	-	4,291,960 ns	-
THREAD 1.1.4	3,435,986,692 ns	49,696,204 ns	-	4,657,710 ns	-
THREAD 1.1.5	4,296,556,041 ns	49,602,527 ns	-	3,389,327 ns	-
THREAD 1.1.6	5,147,248,311 ns	49,584,182 ns	-	3,676,477 ns	-
THREAD 1.1.7	5,992,174,615 ns	49,584,502 ns	-	4,567,889 ns	-
THREAD 1.1.8	6,801,670,398 ns	49,580,671 ns	-	4,926,530 ns	-
Total	30,985,275,728 ns	347,392,016 ns	6,119,340,149 ns	49,953,907 ns	2,315 ns
Average	3,873,159,466 ns	49,627,430.86 ns	6,119,340,149 ns	6,244,238.38 ns	2,315 ns
Maximum	6,801,670,398 ns	49,696,204 ns	6,119,340,149 ns	21,056,930 ns	2,315 ns
Minimum	1,009,489,963 ns	49,580,671 ns	6,119,340,149 ns	3,387,084 ns	2,315 ns
StDev	1,922,131,706.82 ns	47,620.81 ns	0 ns	5,625,811.58 ns	0 ns
Avg/Max	0.57	1.00	1	0.30	1

Figure 13: Paraver trace of Gauss-Seidel with howmanyAux = 1

As we introduced in the beginning of this section, when the value of blocks is too large, there will be more time spent in synchronizations due to the number of dependences. We can see an example of that case by taking a look to the following paraver trace (howmanyAux = 64), where the scheduling and fork/join time has increased in comparison with the one shown in Figure 12.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,634,853,294 ns	-	838,082,317 ns	16,801,448 ns	2,350 ns
THREAD 1.1.2	1,689,357,704 ns	52,378,834 ns	-	2,913,857 ns	-
THREAD 1.1.3	1,853,143,305 ns	52,372,343 ns	-	3,155,442 ns	-
THREAD 1.1.4	2,024,899,141 ns	52,358,918 ns	-	3,982,165 ns	-
THREAD 1.1.5	2,082,129,958 ns	52,330,875 ns	-	2,994,633 ns	-
THREAD 1.1.6	2,133,855,892 ns	52,329,130 ns	-	3,037,574 ns	-
THREAD 1.1.7	2,157,808,566 ns	52,329,122 ns	-	3,036,012 ns	-
THREAD 1.1.8	2,168,257,956 ns	52,326,367 ns	-	2,989,422 ns	-
Total	15,744,305,816 ns	366,425,589 ns	838,082,317 ns	38,910,553 ns	2,350 ns
Average	1,968,038,227 ns	52,346,512.71 ns	838,082,317 ns	4,863,819.12 ns	2,350 ns
Maximum	2,168,257,956 ns	52,378,834 ns	838,082,317 ns	16,801,448 ns	2,350 ns
Minimum	1,634,853,294 ns	52,326,367 ns	838,082,317 ns	2,913,857 ns	2,350 ns
StDev	200,537,459.93 ns	21,114.46 ns	0 ns	4,523,390.23 ns	0 ns
Avg/Max	0.91	1.00	1	0.29	1

Figure 14: Paraver trace of Gauss-Seidel with howmanyAux = 64

5.4 Optional

Implement an alternative parallel version for Gauss-Seidel using #pragma omp task and task dependences to ensure their correct execution. Compare the performance against the #pragma omp for version and reason about the better or worse scalability observed.

R/

Gauss-Seidel using tasks

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey){
        double unew, diff, sum=0.0;
        int howmany=omp_get_max_threads();
        int howmanyAux = howmany;
        char dep[howmany][howmanyAux];
        omp_lock_t lock;
        omp_init_lock(&lock);
        #pragma omp parallel
        #pragma omp single
        for (int blockid = 0; blockid < howmany; ++blockid) {
          int i_start = lowerb(blockid, howmany, sizex);
          int i_end = upperb(blockid, howmany, sizex);
```

```
            for (int z = 0; z < howmanyAux; z++) {
              int j_start = lowerb(z, howmanyAux, sizey);
              int j_end = upperb(z,howmanyAux, sizey);
              #pragma omp task firstprivate (j_start,j_end, i_start, i_end)
             depend(in: dep[max(blockid-1,0)][z], dep[blockid][max(0,z-1)])
             depend (out: dep[blockid][z]) private(diff,unew)
             {
                double sum2 = 0.0;
                for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                  for (int j = max(1, j_start); j<= min(j_end, 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];
                   sum2 += diff * diff;
                   u[i*sizey+j]=unew;
                   }
                }
               omp_set_lock(&lock);
               sum += sum2;
               omp_unset_lock(&lock);
            }
           }
         }
         omp_destroy_lock(&lock);
         return sum;
}
```

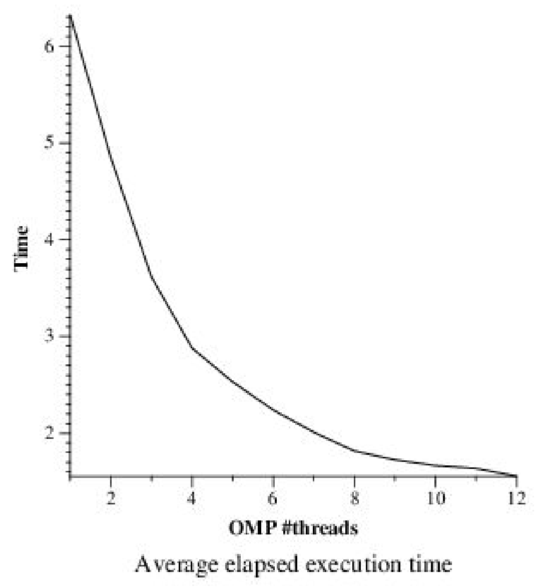
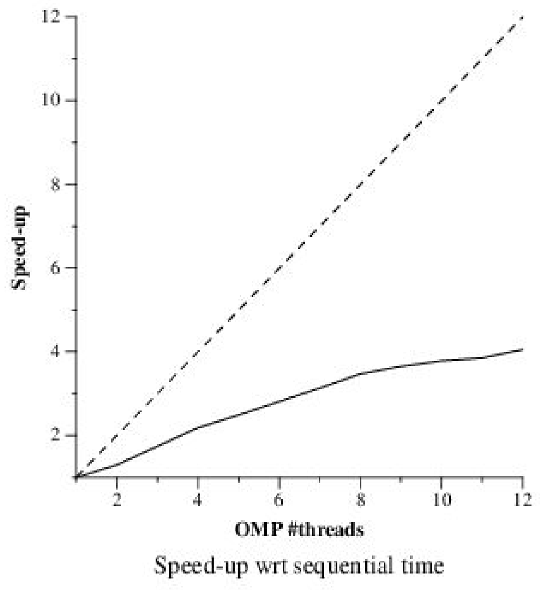
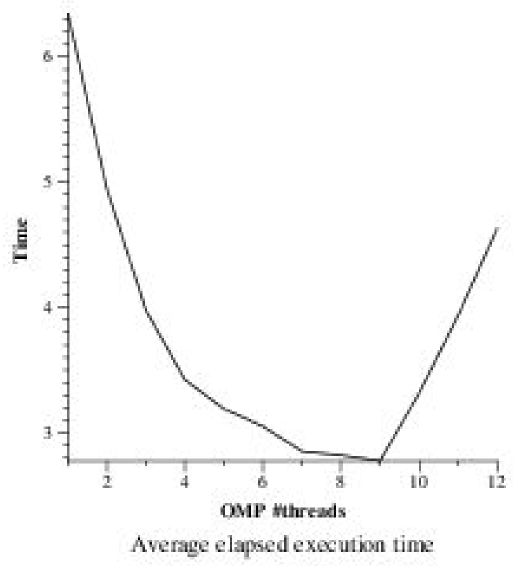
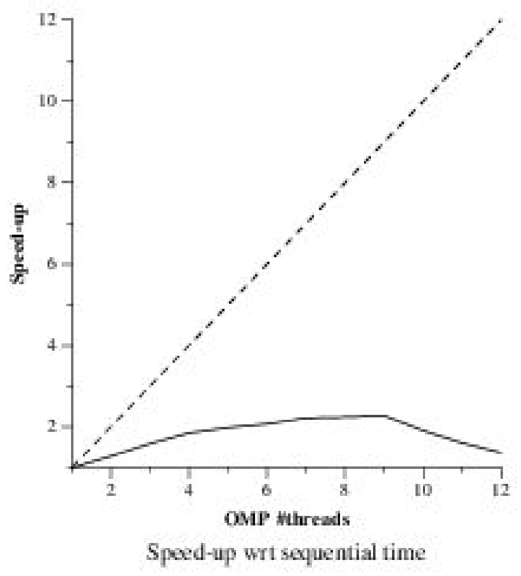
Strong-omp.sh plots of Gauss-Siedel parallelitzation	**Strong-omp.sh plots of Gauss-Siedel parallelitzation with task implementation**
 Average elapsed execution time  Speed-up wrt sequential time	 Average elapsed execution time  Speed-up wrt sequential time

Table 6: Plot with the comparison between Gauss-Siedel with and without tasks

As we see can see in Table 6 using a task strategy is worse than using the omp for strategy. This is basically due to the additional time that is spent managing tasks that

makes our performance slower at any number of threads.
We can also see that once we reach 9 threads in the task strategy, the performance will go inversely proportional to the number of threads because the managing tasks time will be proportional to the number of threads.

Annex

The following results have been obtained to generate figure 11 where we want to obtain the optimal howmanyAux value for the ratio computation/synchronization in the parallelization of Gauss-Seidel 8 threads.
We had no idea about the optimal value so at first we thought it would be a lower value which explains that we've trying consecutive values until 20.

howmanyAux	Execution time (seconds)
1	7.061
2	4.142
3	3.146
4	2.652
5	2.363
6	2.191
7	2.215
8	1.978
9	1.964
10	1.873
11	1.930
12	1.748
13	1.819
14	1.741
15	1.782
16	1.675
17	1.783
18	1.783
19	1.746
20	1.802
25	1.686

32	1.477
40	1.568
64	2.383