5th deliverable
Geometric (data) decomposition
Solving the heat equation

Omair Iqbal
Group 11
Par1103
May 2016

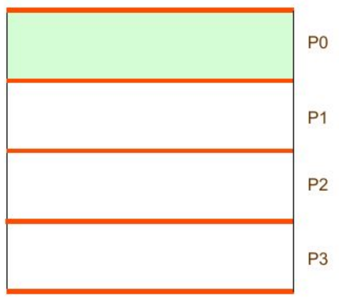
Introduction

In this lab deliverable I am going to parallelize a sequential code (heat.c) that simulates heat diffusion in a solid body using two different solvers for the heat equation (Jacobi and Gauss-Seidel).
The program is executed with a configuration file (test.dat) that specifies the maximum number of simulation steps (iterations), the size of the body (resolution), the solver to be used (Algorithm) and the heat sources, their position, size and temperature. The program generates performance measurements and a file heat.ppm providing the solution as image (as portable pixmap file format), a gradient from red (hot) to dark blue (cold).

Parallelization strategies

Block data decomposition by rows is the main parallelization strategy carried out which divides the rows of the matrix and gives to each processor a certain number of it, so we can take advantage of each thread working in its maximum load.

In the picture in the right we can see that the rows of the matrix are spread among the 4 threads so each one can work independently (we also have to be careful with the dependences).



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?

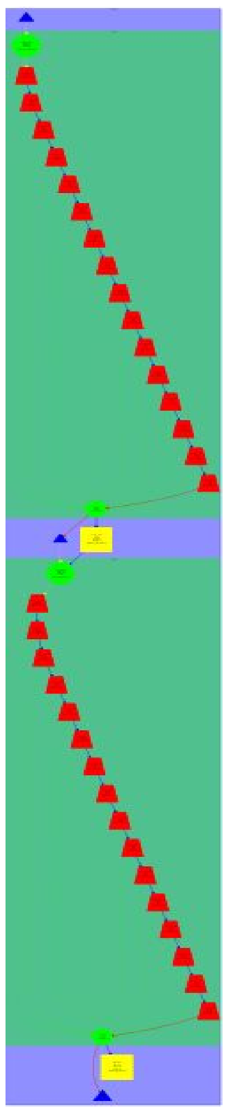
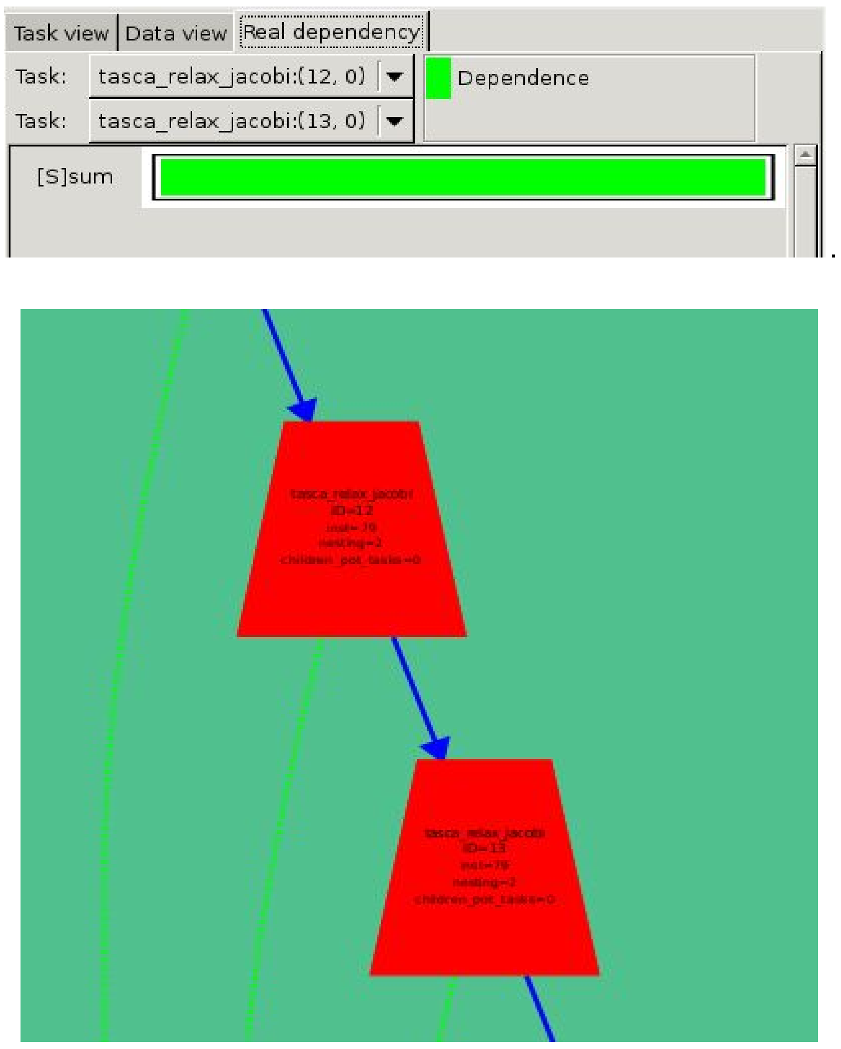


Figure 1: Task Graph generated by Jacobi



Task: tasca_relax_jacobi:(12, 0)
Task: tasca_relax_jacobi:(13, 0)
[S]sum

Figure 2: Jacobi data dependences, in the first picture we can see that the sum variable is causing the dependence.

As we can see in the dependence task graph above (Figure 1), we have a dependence that prevent us from parallelizing the code, the variable "sum", so we are going to disabled it and then generate the task graph again to see the potentially parallelizable tasks.

```
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;
}
```

Figure 3: Jacobi instrumentation with the variable sum disabled.

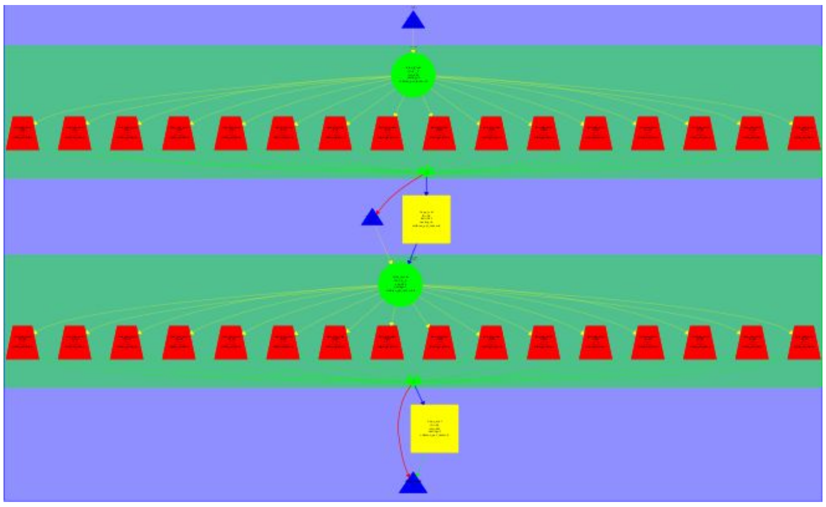


Figure 4: Task graph generated by Jacobi disabling the variable sum and showing the potentially parallelizable tasks.

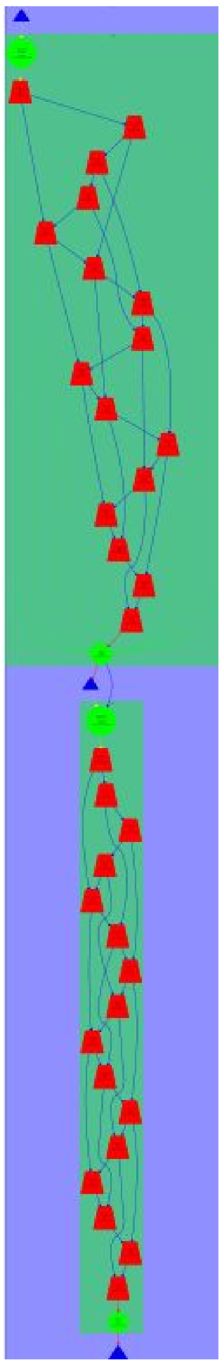
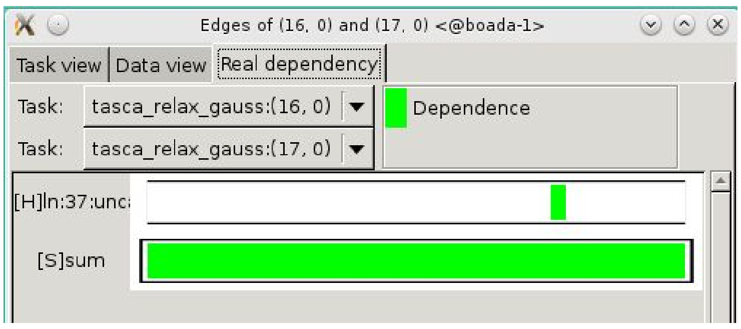


Figure 5: Task Graph generated by Gauss-Seidel



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

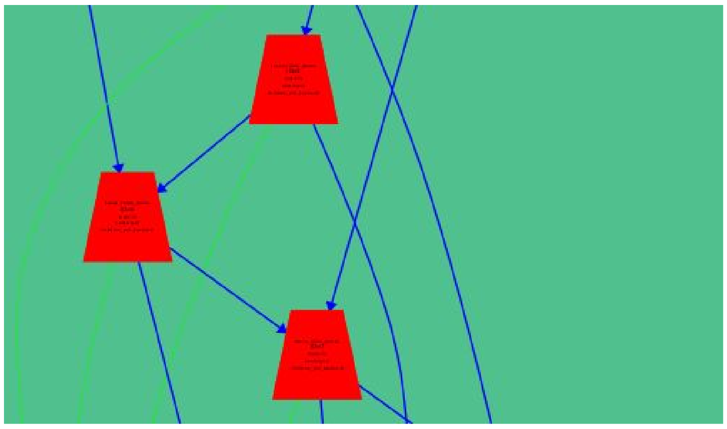


Figure 6: Gauss-Seidel data dependences, in the first picture we can see that the sum variable is causing the dependence alongside with another one

As we can see in the first picture of the figure 6 more than one variable are causing causing dependences. 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. So we are going to make the necessary changes in the code and generate the respective task graph again.

```
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;
}
```

Figure 7: Gauss instrumentation with the variable sum disabled.

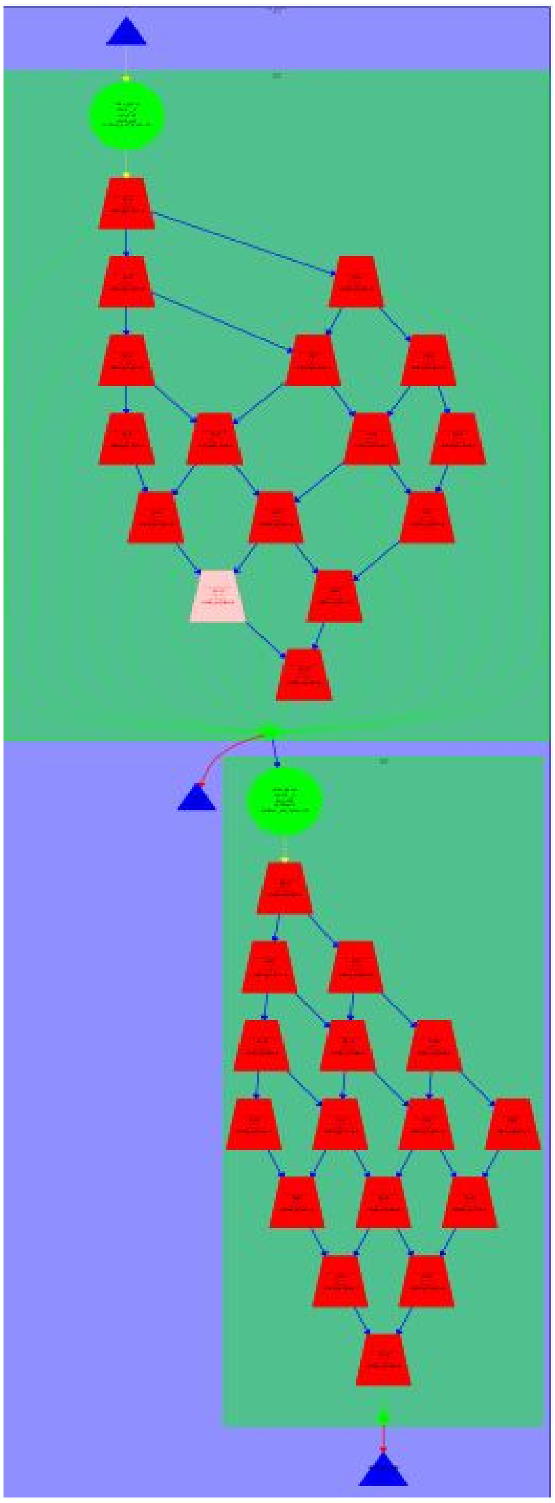


Figure 8: Task Graph generated by Gauss-Seidel after disabling the variable sum and showing the potentially parallelizable tasks.

We will protect them in the parallel code by adding the directive **#pragma omp parallel for reduction(+:sum) private(diff)** in the both cases.

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.

The data decomposition strategy applied to solve the problem is a Block Data Decomposition which one is explained in the “Parallelization strategies” part of this document.

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.

```
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;
}
```

Figure 9: Jacobi instrumentation parallelized.

```
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 ];
}
```

Figure 10: copy_mat function of the solver-omp.c file parallelized.

To parallelize the Jacobi function we have added pragma directive shown in bold in the figure 9. With the for clause the loop is executed by multiple threads at the same time because we want to implement block data decomposition.
The variable sum calculates the final result, but as the loop is executed by multiple threads each one will calculate its own partial result, so we want to add all of them in the final, avoid any data race condition and get to total result that's why we have added the reduction clause.
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.
We also decided to parallelize the copy_mat function shown in the Figure 10 adding the pragma directive shown in bold in the same figure.

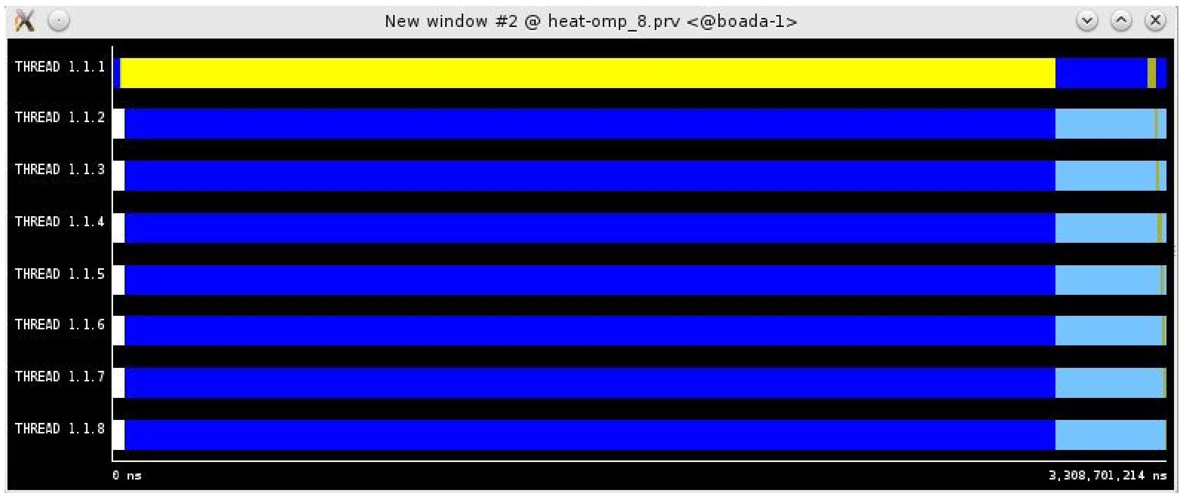


Figure 11: Paraver trace of Jacobi intrumentation parallelized

It can be seen in the paraver trace (figure 11) 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.

```
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;
}
```

Figure 12: Jacobi instrumentation parallelized with and optimization to correct the load balance between threads.

We are assigning the maximum number of threads available by calling the function in bold in the figure 12, so when we are distributing in blocks all the work we will consider the maximum number of threads available, this way all the workloads of each thread will be more balanced.

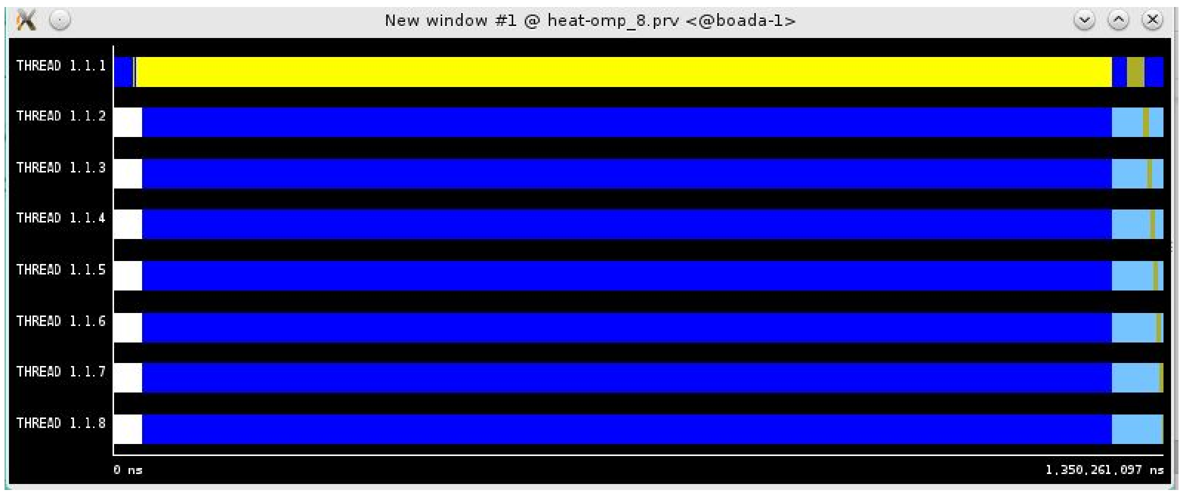


Figure 13: Paraver trace of Jacobi instrumentation parallelized with code optimization

As we can see in the in the Paraver trace of the figure 13 the time execution and work balance are way better.

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.

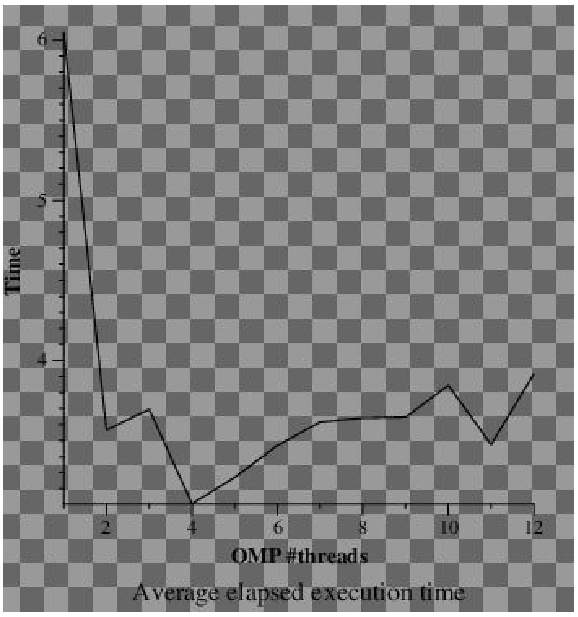
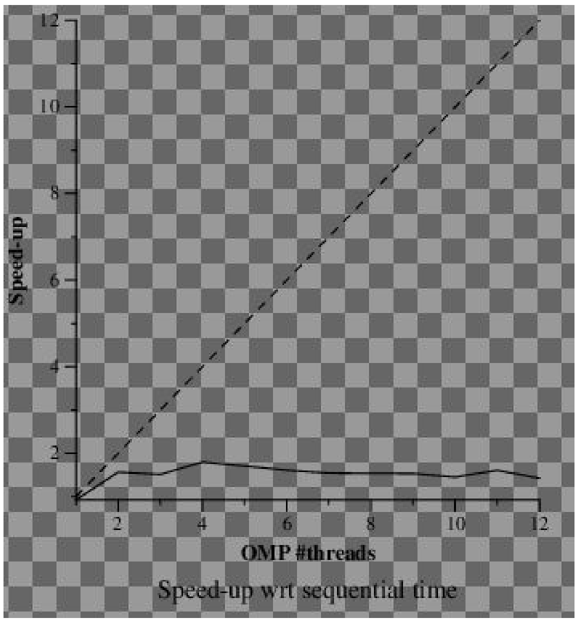
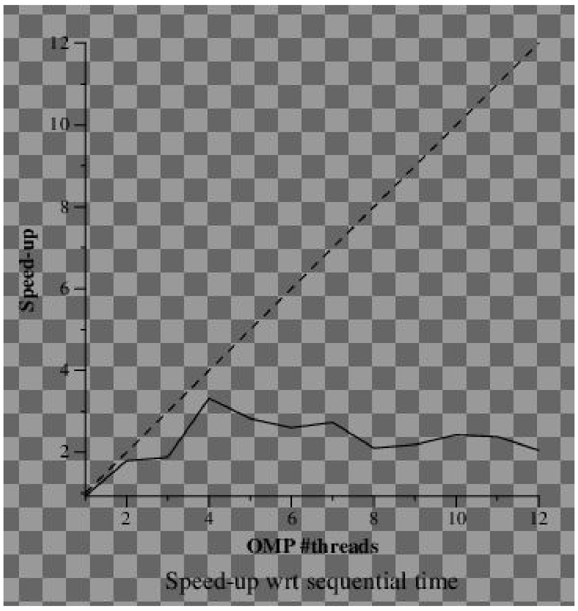
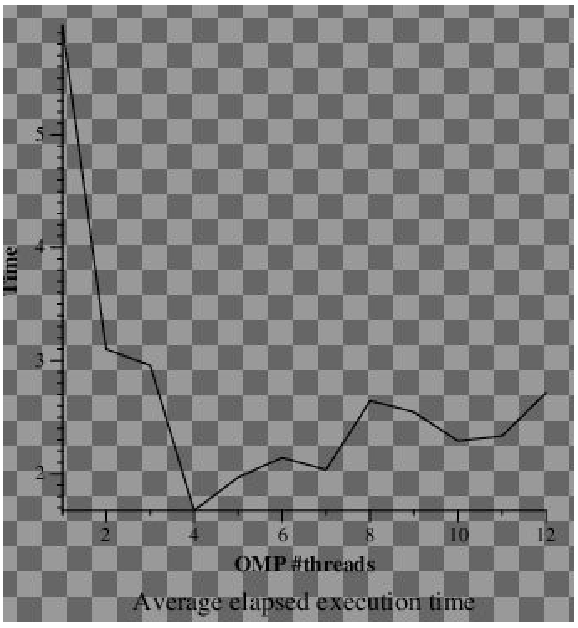
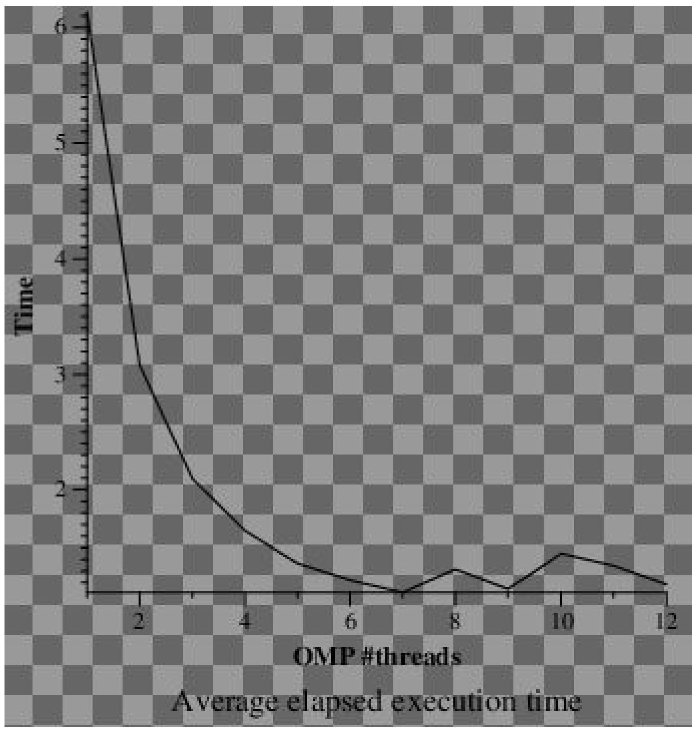
 	 

Figure 14: Strong-omp.sh plots of Jacobi parallelization without copy_mat function parallelizated	**Figure 15:** Strong-omp.sh plots of Jacobi parallelization with copy_mat function parallelizated

As we can see in Figure 15, the performance is not as good as we thought it would be by parallelizing the first loop. The speedup is far from being adjust to the ideal curve once we reach 4 threads, so after looking in the Paraver trace and to locate the problem (load balance) a optimization has been done.



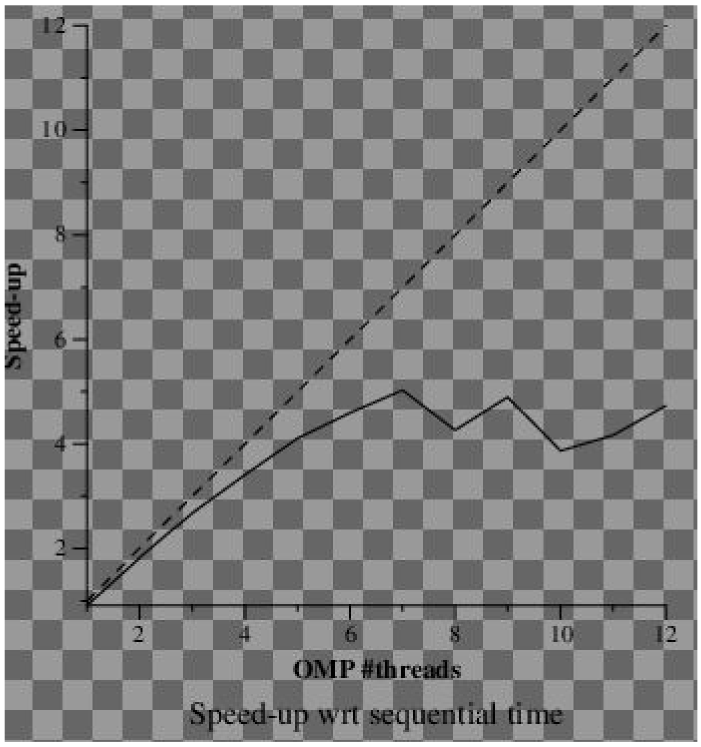


Figure 16: Strong-omp.sh plots of Jacobi parallelization with copy_mat function parallelizated and the extra optimization

As we can see our speed-up plot (figure 16) is now much better than the one seen in figure 15 (especially in the results for more than 4 threads).

OpenMP parallelization and execution analysis: Gauss-Seidel

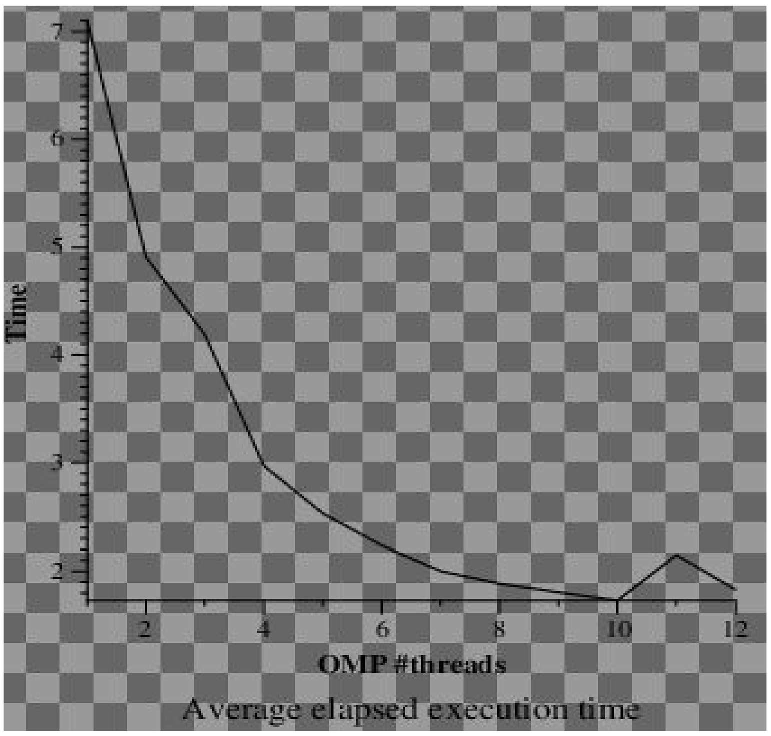
1. Include the relevant portions of the parallel code that implements the Gauss-Seidel solver, commenting how you implemented the synchronization between threads

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
        double unew, diff, sum=0.0;
        int howmany=omp_get_max_threads();
        int howmanyAux = howmany;
        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;
}
```

Figure 17: Gauss-Seidel instrumentation with the right openMP to parallelize the code.

The idea is to divide the rows on the threads in order to improve the performance and the execution time. The value that will determine the division of blocks is the variable howmanyAux, that in this case will be fixed with the howmany value.
We have top dependences and left dependences as shown in figure 8, we don't have to worry about the left one because each block is executed sequentially, so every time we will go for the next block we will already have the result of the previous one. The other dependency is more difficult to treat, so we have 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”), by this way the “depending” threads will be waiting until their dependencies are calculated.
A new variable “unew” has been added to, which represent the auxiliary matrix, to the private clause since it is used to calculate the final result and we need to avoid a data race condition once again.
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.



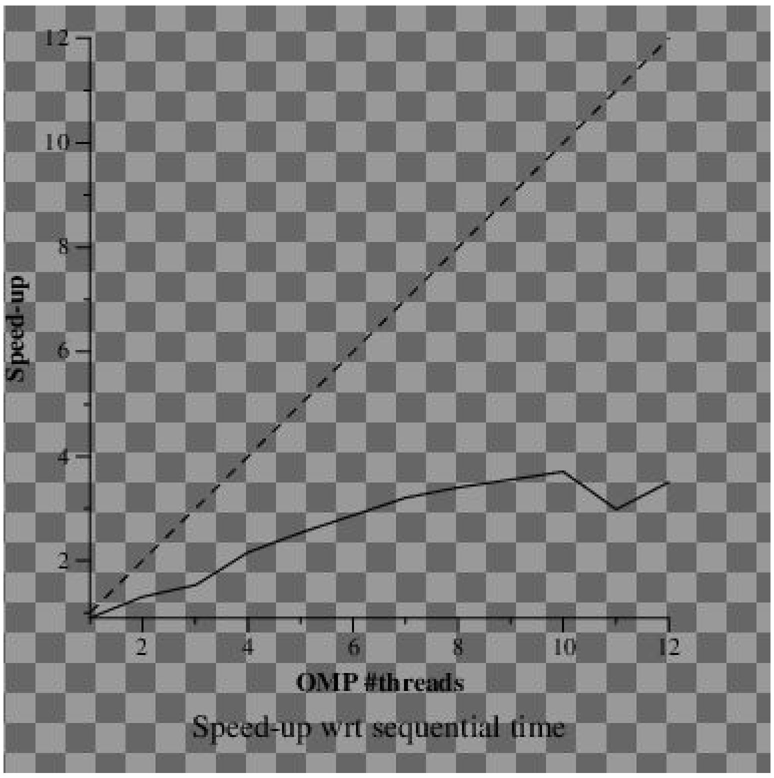


Figure 18: Strong-omp.sh plots of Gauss-Seidel parallelization with copy_mat function parallelized.

As we can see in the figure 18 this algorithm is worst than the Jacobi because the main problem is the waiting time that the last thread has due to the dependences mentioned before.

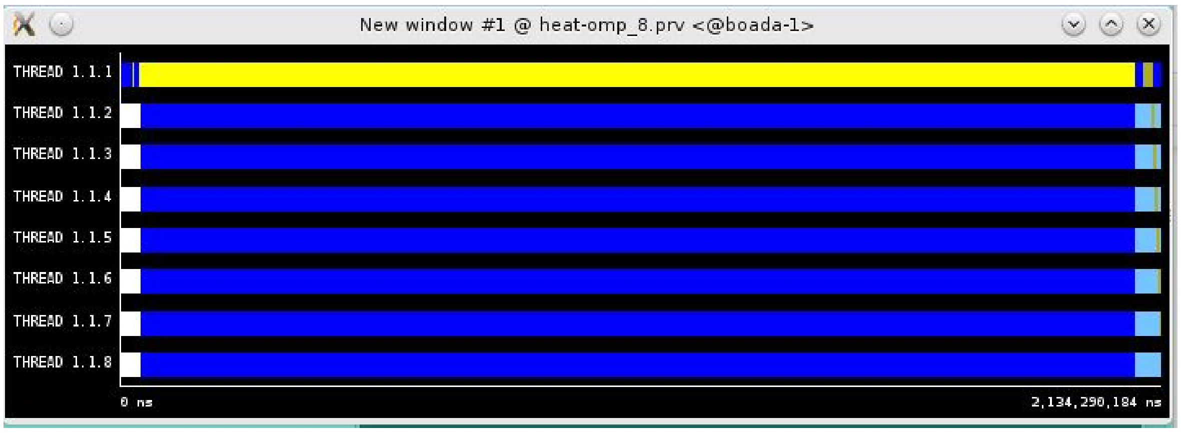


Figure 19: Paraver trace of Gauss-Seidel with howmanyAux = howmany

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. 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.

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**.

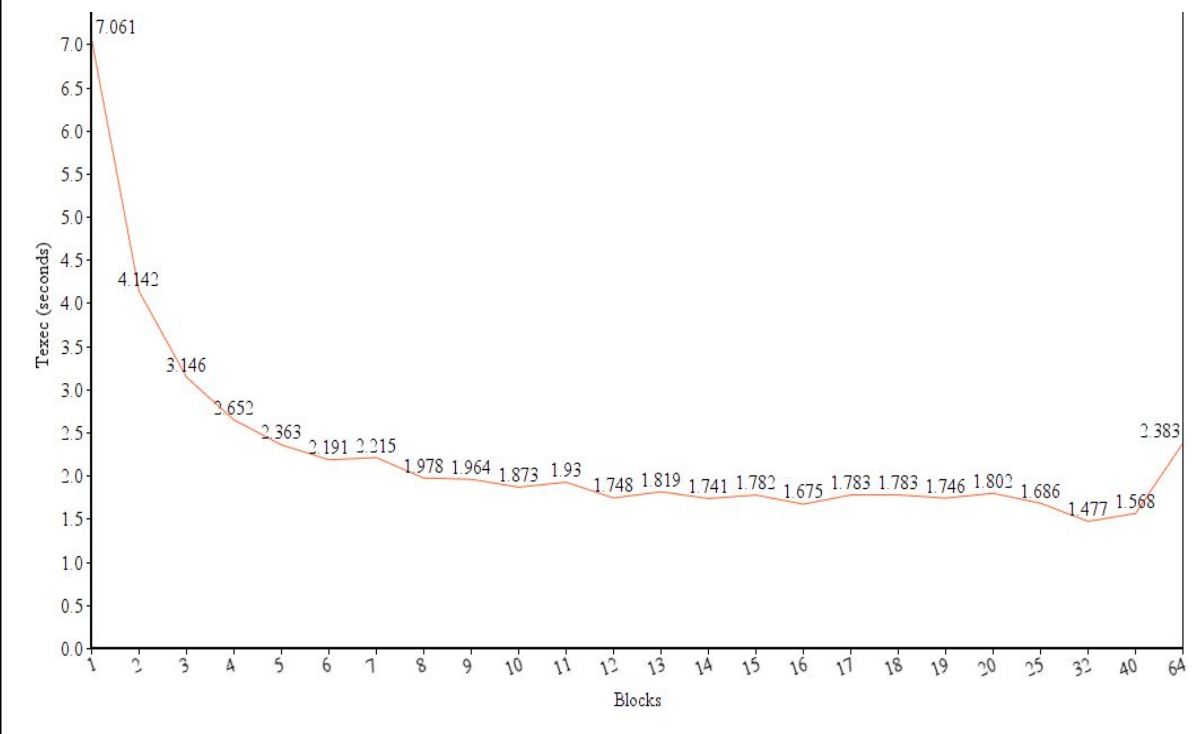


Figure 20: Graphic generated with the values

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 (figure 19).

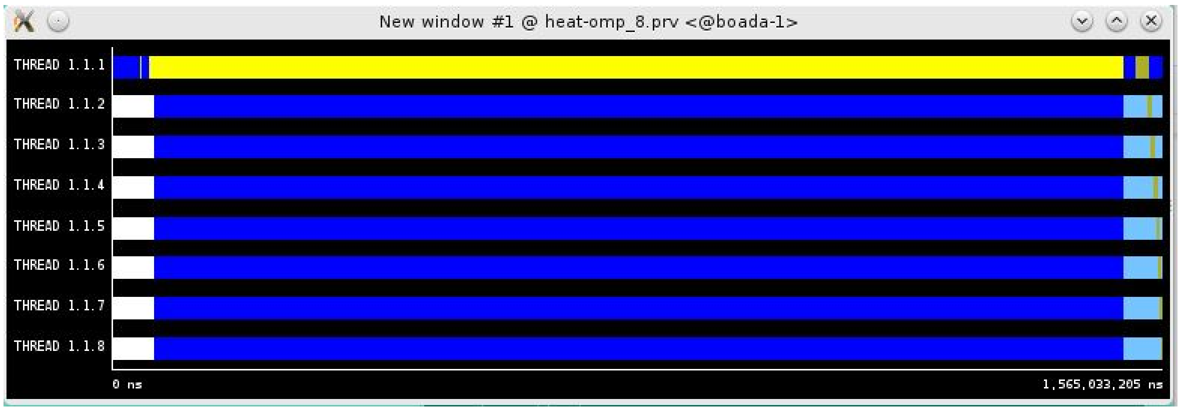
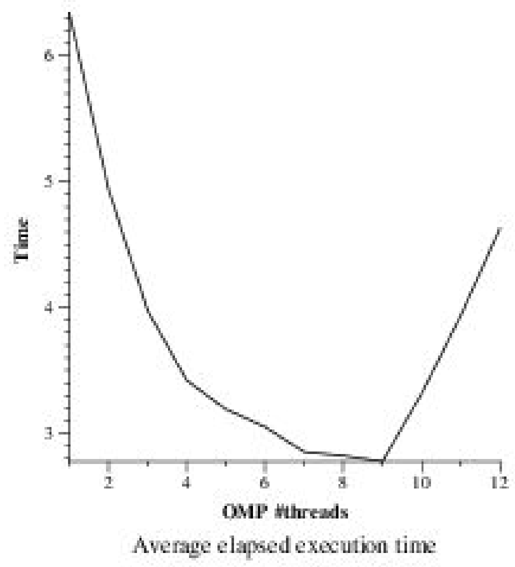


Figure 21: Paraver trace of Gauss-Seidel with howmanyAux = 32.

Optional 1: 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.

```
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;
}
```

Figure 22: Gauss-Seidel instrumentation using tasks.



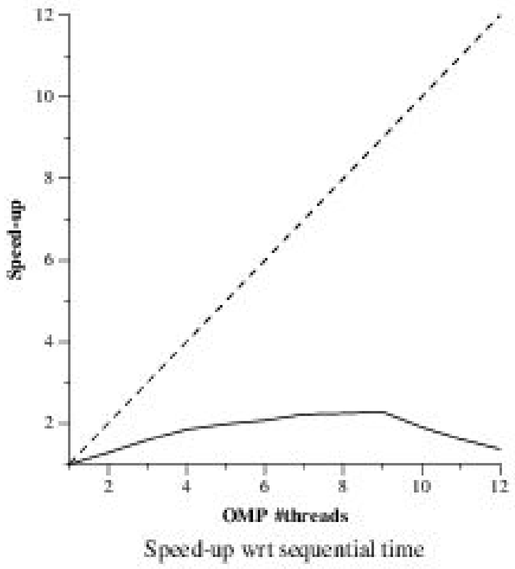


Figure 23: Strong-omp.sh plots of Gauss-Siedel parallelitzation with task implementation

As we see can see in figure 23 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.

Conclusions

The main conclusion we can get from this lab session is that the block data decomposition has a lot of problems and it's difficult to implement in programs where there are a lot of dependences and probably there are better strategies to solve this kind of problems.