**PAR Laboratory Assignment Lab 5: Geometric (data) decomposition: solving the heat equation**

Oscar Faixat Llanas

Xavier Algarra Torelló

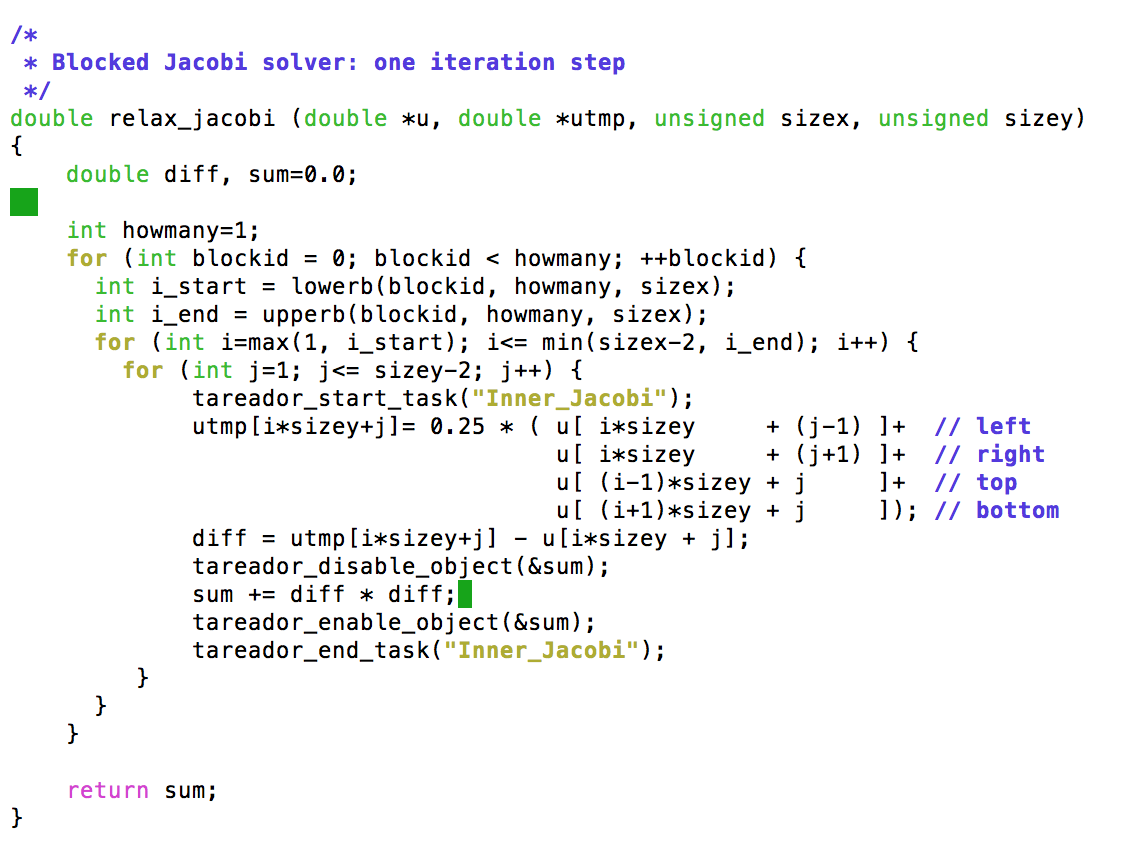
Grup: PAR1212

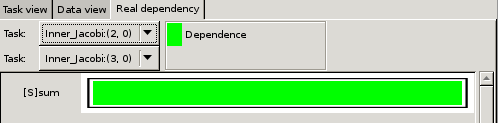
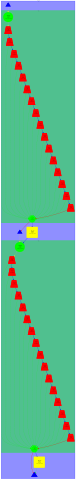
PAR 2017Q2

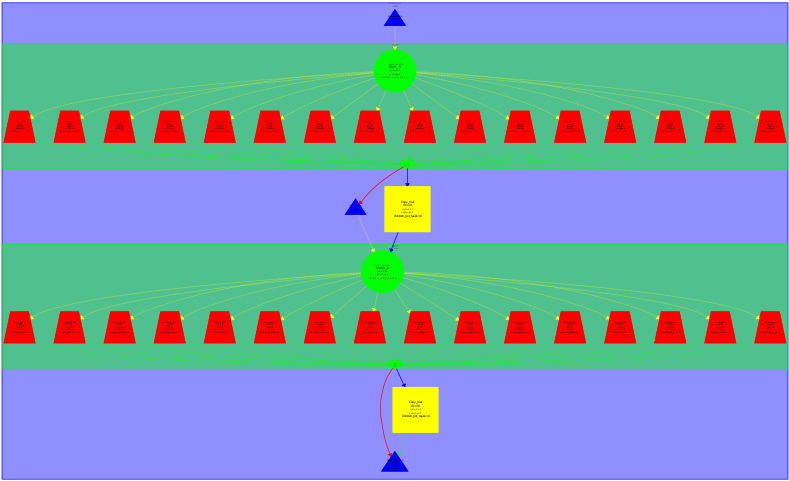
18/05/2017

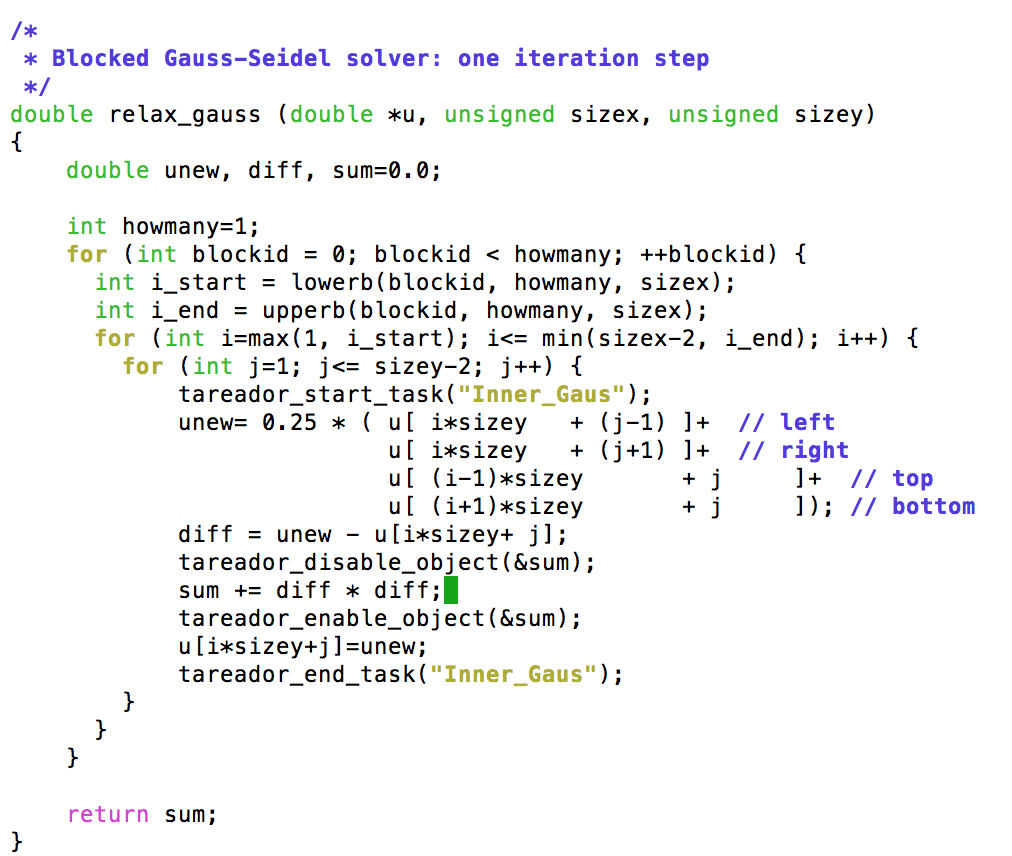
**5.1 Analysis with Tareador**

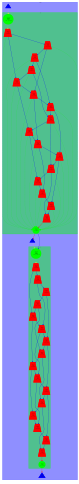
1. **Include the relevant parts of the modified solver-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear in the two solvers: Jacobi and Gauss-Seidel. How will you protect them in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

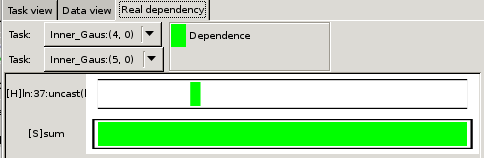
****



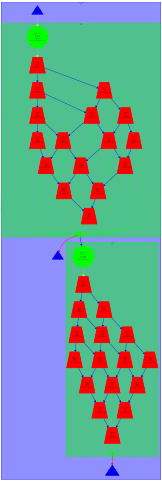
We can see from the screenshots above that in the relax\_jacobi function that the accesses to the variable sum is serializing the tasks. In order to fix this, we should use the pragma omp for with the reduction clause (+:sum). With this clause we will be assured that the variable sum will be private for each thread and once all of the tasks are finished, all the different values for sum will be added. The screenshot below is the resulting task graph when we disable the sum variable in the tareador api. 

****



****

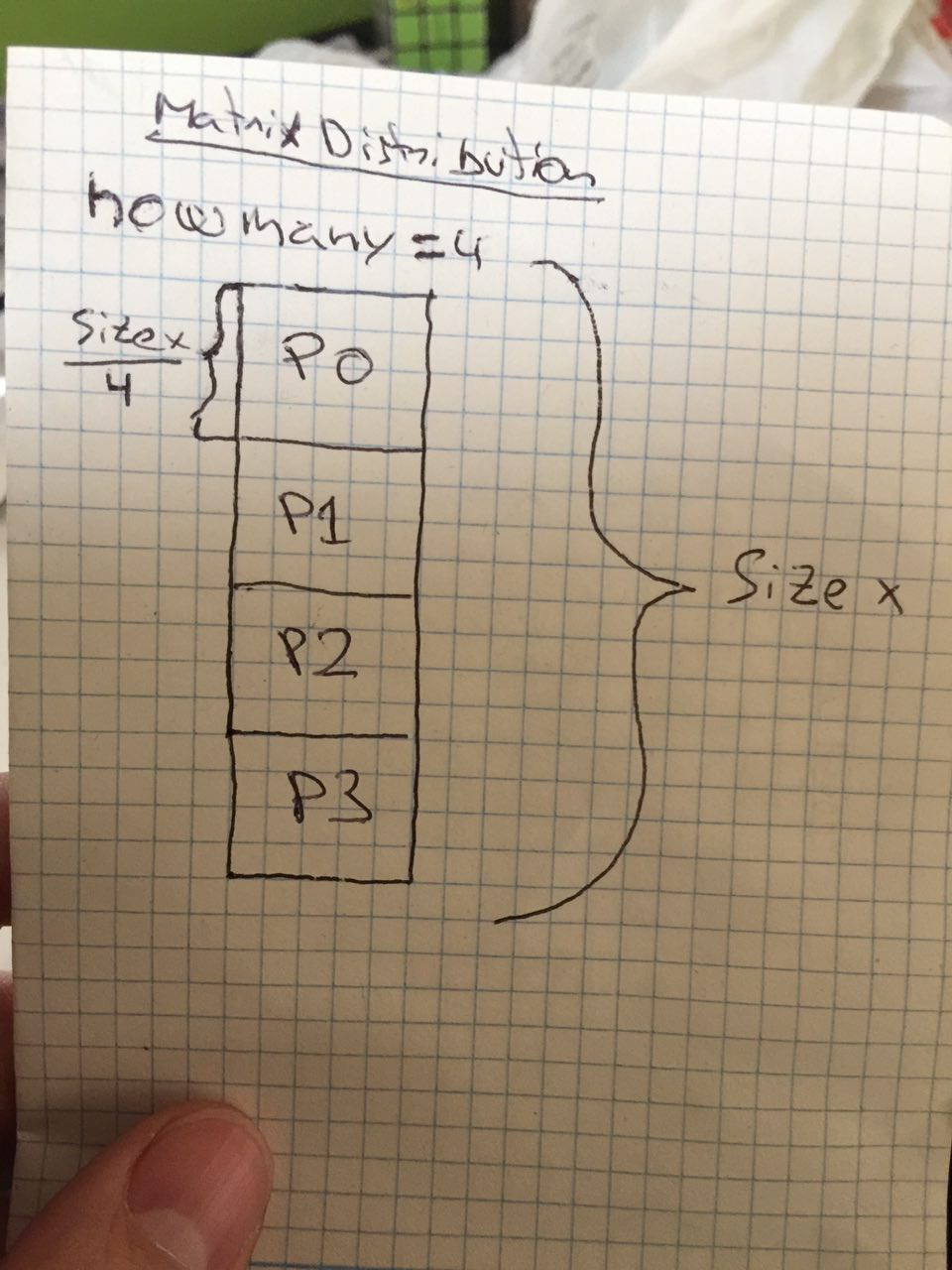
The screenshot above, shows us that in the relax\_gauss function there are dependencies in the upper level and to the left. To solve this, we need to divide the rows that have been assigned to each thread in various blocks of columns. By doing this we will be able to achieve a better parallelism. The screenshot below is the task dependency graph of the Gauss-Siedel function after disabling the variable sum.

****

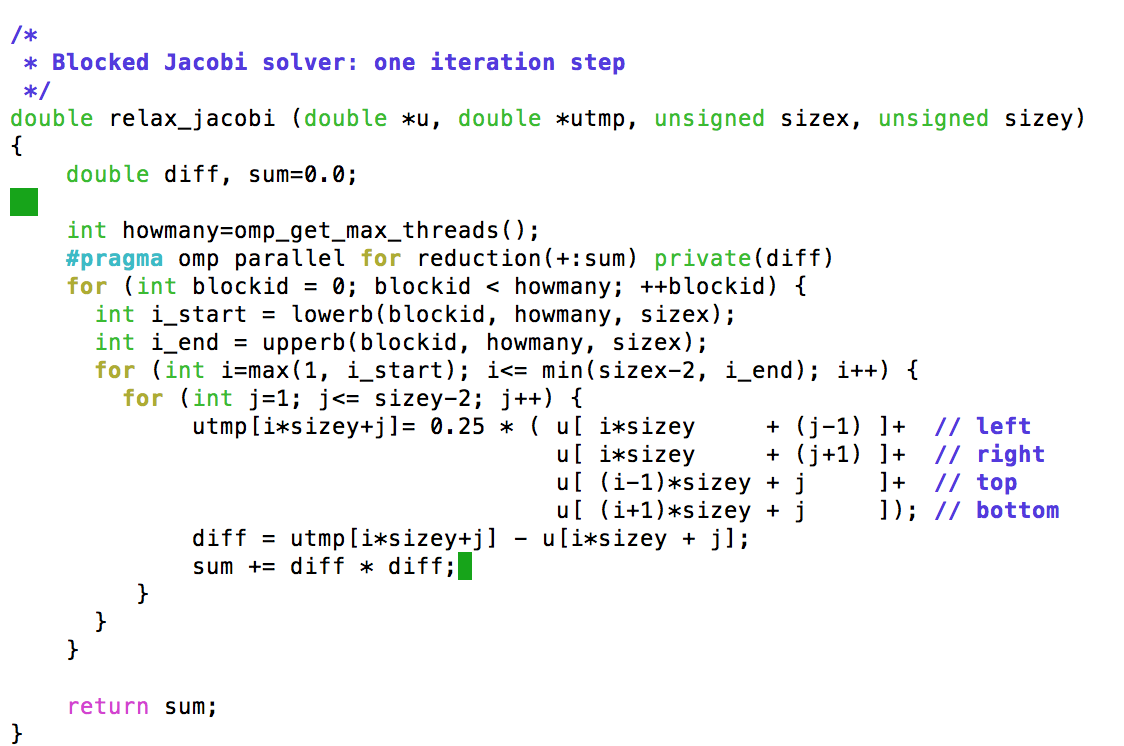
**5.2 OpenMP parallelization and execution analysis: Jacobi**

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.**

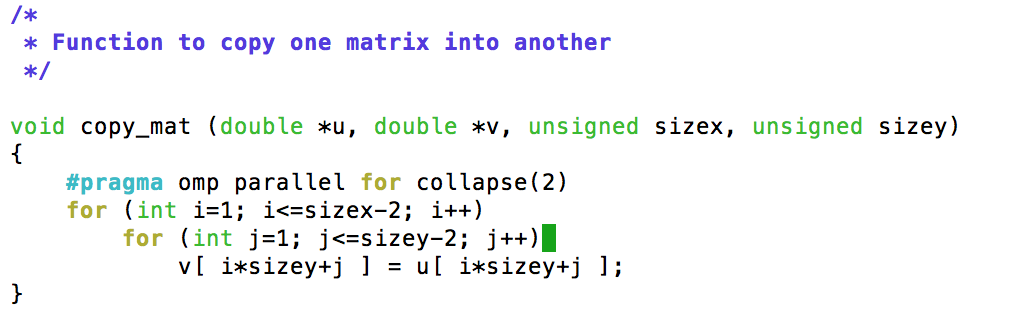
The data decomposition strategy that we have used to solve the problem is to divide the matrix into blocks. We do this in the relax\_jacobi function. Inside the first for loop i\_start and i\_end are calculated from scratch and the idea is to assign a block to each in order to compute the utmp matrix with the use of the matrix u values. To calculate the values o i\_start and i\_end the program uses the functions lowerb and upperb. The purpose of these functions is to return the limits of the block that the thread executing it has to compute. We think that this is a Geometric Block Data Decomposition.

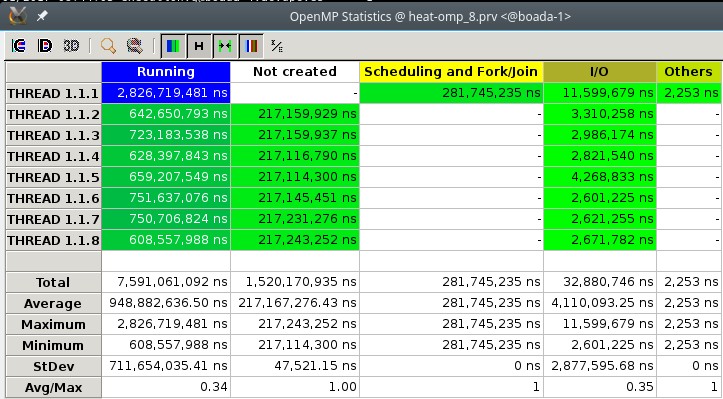


1. **Include the relevant portions of the parallel code that you implemented to solve the heat equation using the Jacobi solver, commenting whatever necessary. Including captures of Paraver windows to justify your explanations and the differences observed in the execution.**

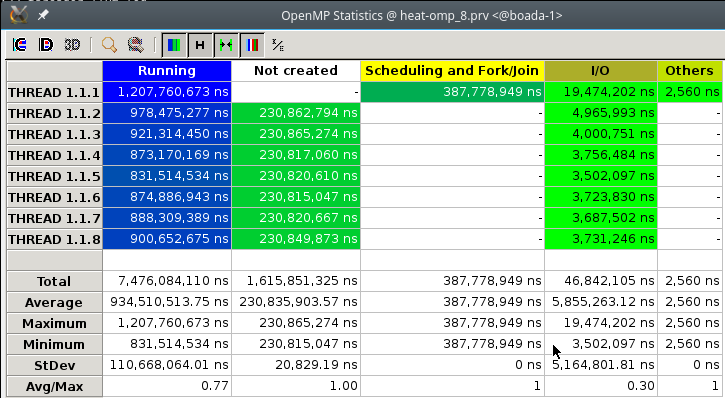
****

The screenshot above, belongs to the code for the relax\_jacobi function. To parallelize the function we make use of the pragma omp for to distribute the work to all of the threads. As we mentioned in the tareador exercise, we need to use the reduction clause on the variable sum so that when all of the threads finish we obtain the total sum. We have also found it necessary to make the diff variable private as it was declared before the pragma was instantiated and so it will be shared by default. If we make it stay as a shared variable a data race condition will occur and we want to avoid that.

****

****

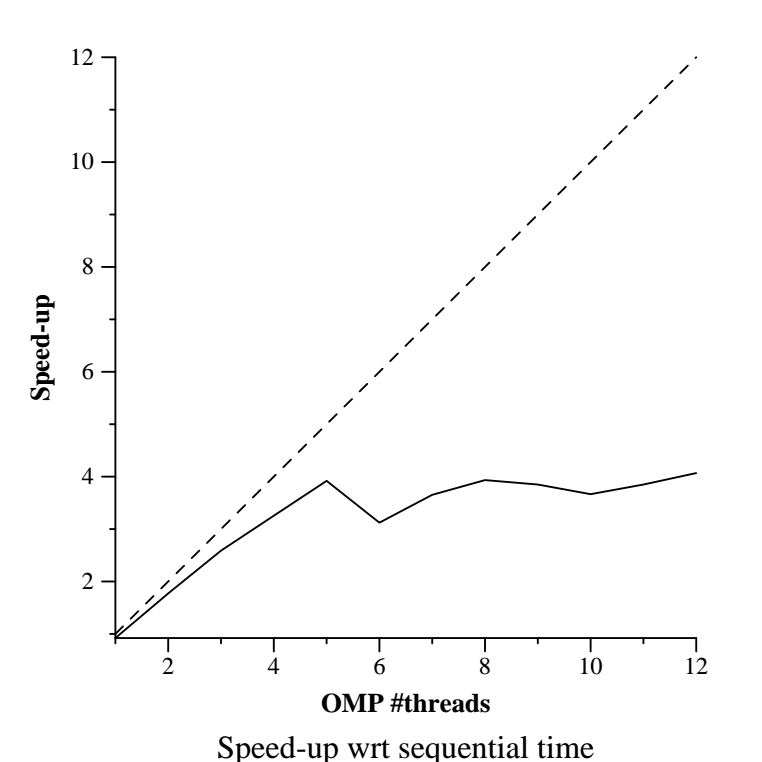
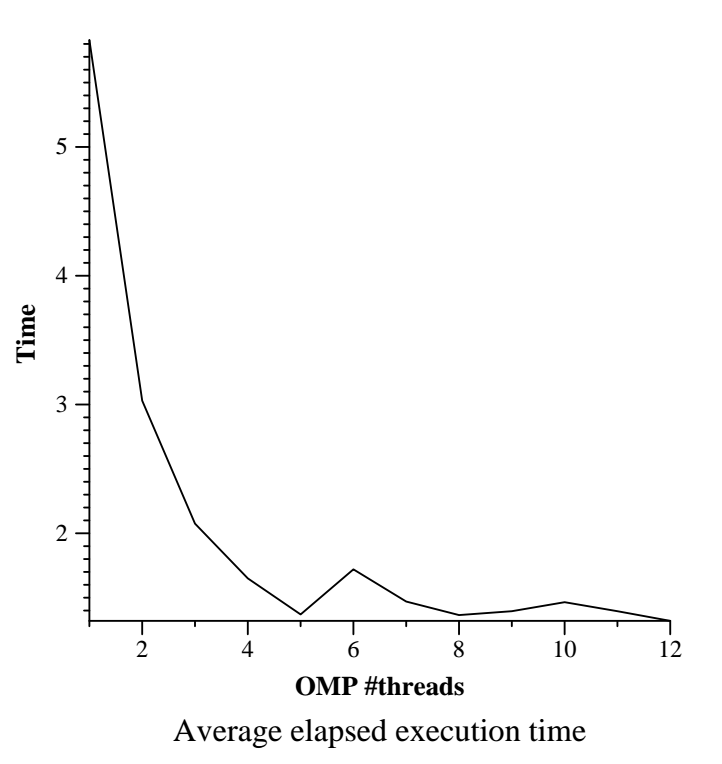
**8-threads without copymat parallelized**



**8-threads with copymat parallelized**

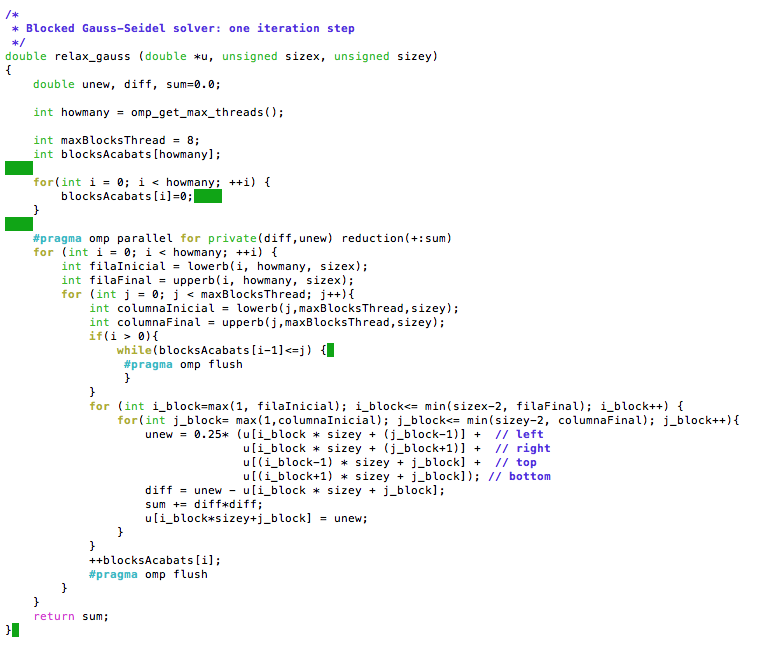
The paraver traces above, show us that parallelizing the copy\_mat function considerably reduces the execution time. In order to parallelize the function we used a simple pragma omp parallel for collapse(2) clause as there is a nested loop. Parallelizing this function is relatively simple as there are no dependencies that we have to worry about.

**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.**



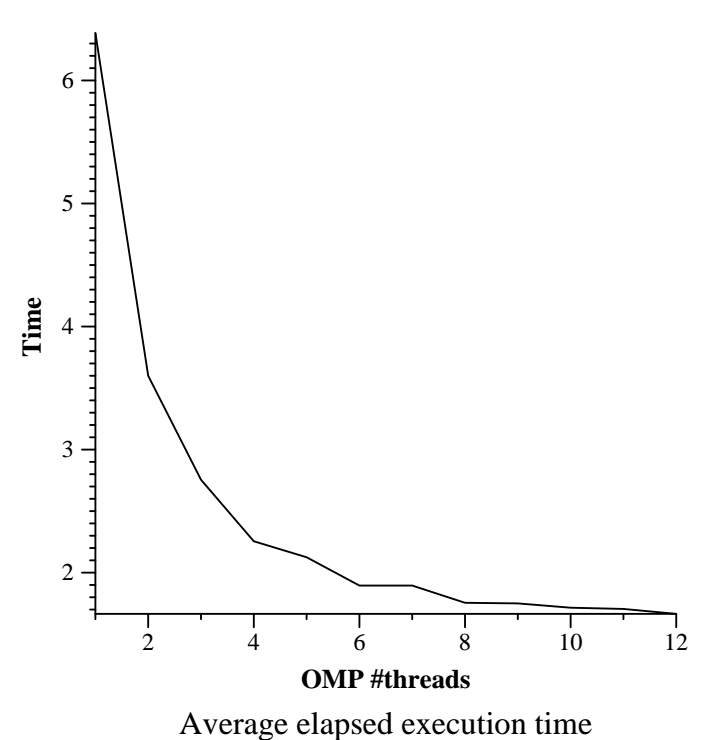
As we can observe from the speedup above, there is a good increase in speedup up to 5 threads. After that, the speedup stabilizes at around 4. It is worth mentioning that this is not the speedup that we always get, there are different variations and some have worse results than what we see here. We have observed after running several iterations that this is one of the most common speedups we obtain.

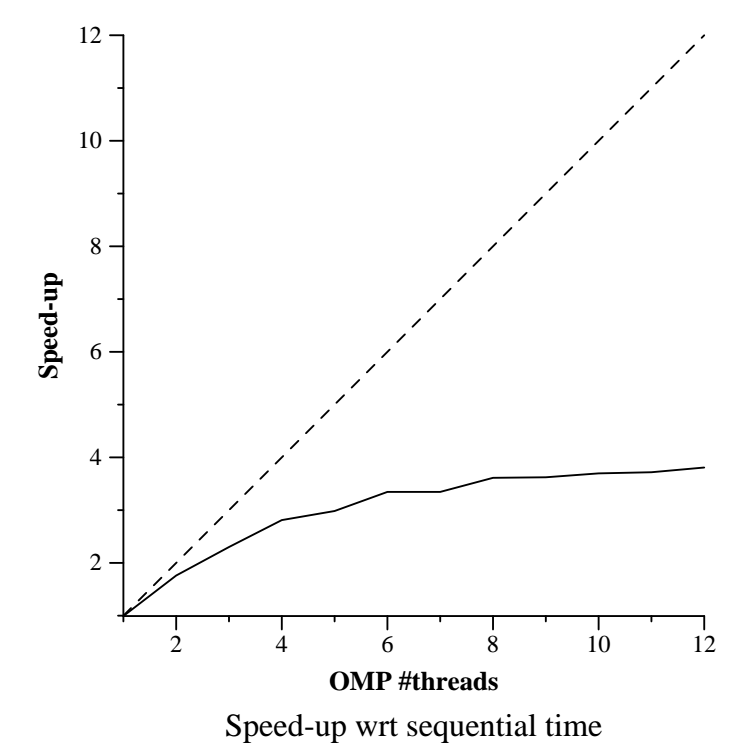
**5.3 OpenMP parallelization and execution analysis: Gauss-Seidel**

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

The idea is to divide the rows per thread into blocks. The amount of blocks may affect the performance of the program as having too many blocks will cause synchronizations and having too little the threads will spend too much time waiting. Since the blocks are being executed sequentially, we shouldn’t need to worry about the left dependency. Therefore, we only need to guarantee the top dependency. In order to accomplish this, we will make use of a vector that controls which blocks are finished (blocksAcabats).

1. **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.**





As we can see above, the speedup is somewhat similar to that of the Jacobi, however to achieve the same amount of speed-up that with jacobi can be obtained with 5 threads with gauss we require 10-12 threads. Also the execution time is still lower with jacobi than it is with gauss. In order to improve we need to find the suitable number of blocks to balance the cost of synchronizations and the waiting time. Given these results we conclude that the Jacobi function be parallelized much better since it doesn’t have dependency between its elements as it uses an auxiliary matrix.

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

As we explained before, if the value is too small there is going to be too much time spent waiting and if its too big, a lot of time will be spent with synchronizations. After testing the algorithm with different values we have found that 8, 16 and 32 are the best values. If we go bigger, the amount spent with synchronizations is way too high.