Laboratory Assignment 5

PAR - Q2 2019/2020

Marc Monfort Grau
Albert Mercadé Plasencia
Group par2302

May 7th 2020

Index

Introduction

In this report, we analyse and study the parallelization of the sequential code that simulates head diffusion in a solid body using the Jacobi and Gauss-Seidel solvers for the heat equation. Using *Tareador* we investigate the potential parallelism for both solvers with respect to the task size, from finer- to coarser-grain task decompositions. Then we parallelize both algorithms using OpenMP and evaluate their performance.

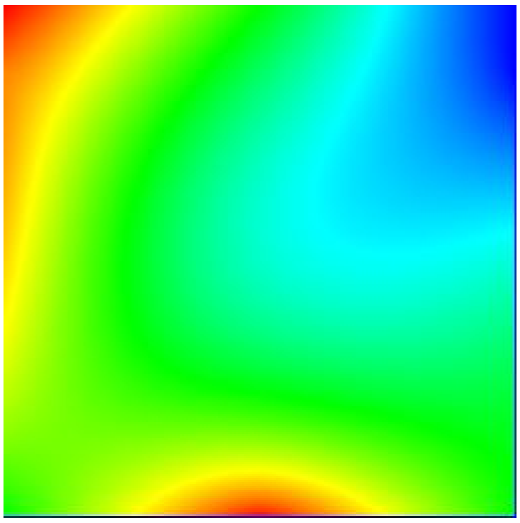


Figure 1. heat-jacobi.ppm

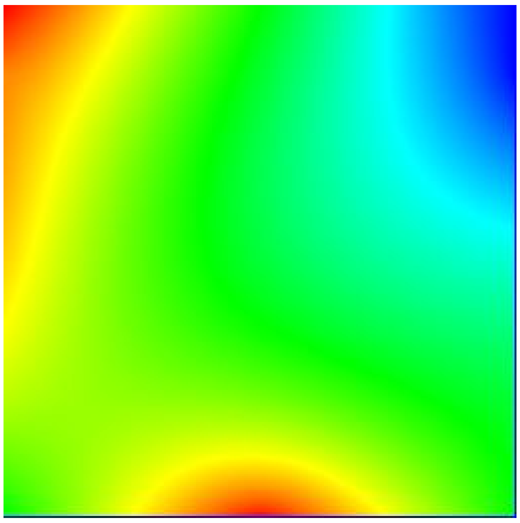


Figure 2. heat-gauss-seidel.ppm

Tareador analysis

In this section of the report we aim to evaluate the potential parallelism of the Jacobi and Gauss-Seidel solvers for the heat equation. We will look at what grainsize in the task decomposition provides us with the best overall performance.

Jacobi

In this initial analysis, we are looking at a coarse-grain task decomposition by only creating a task for every call to `relax_jacobi` and `copy_mat`. The resulting task decomposition graph can be observed below in Figure 3.

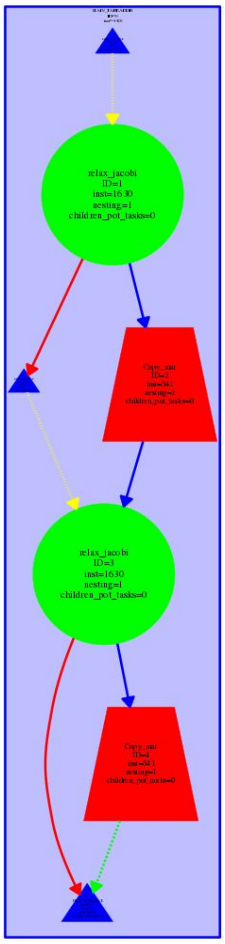
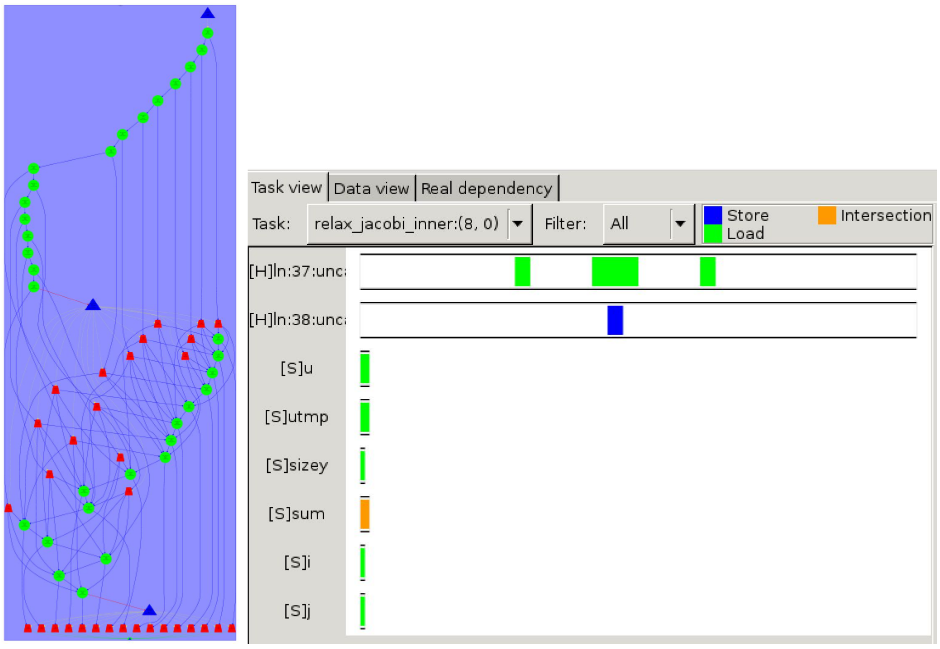


Figure 3. Tareador decomposition for the initial Jacobi version

We now reverse our approach and attempt a much finer-grain task decomposition by creating a task for every iteration of the innermost loop in the `relax_jacobi` function. We do the same for the `copy_mat` method as well. The resulting *Tareador* task decomposition can be observed below in Figure 4 as well as the Dataview for one of the nodes, which shows that the serialization of the tasks is caused by the variable `sum` that is keeping the partial results for every iteration, as it is not only being read but also written to.



Task: relax_jacobi_inner:(8, 0)
Filter: All

Figure 4. Finer-grain Tareador decomposition for Jacobi and Dataview for one of the nodes

To temporarily filter the analysis of `sum` and explore a possible increase in parallelism, we have made use of the `tareador_disable/enable_object(...)` method, as we can observe below in Code 1.

```
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_mat_inner");
           v[ i*sizey+j ] = u[ i*sizey+j ];
       tareador_end_task("Copy_mat_inner");
 }}

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("relax_jacobi_inner");
        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("relax_jacobi_inner");
   }}}
   return sum;
}
```

Code 1. Finer-grain Jacobi with variable `sum` ***protected***

In Figure 5 we can see the task decomposition graph obtained when filtering sum and it is clear that we observe a significant improvement in the parallelisation of the Jacobi solver. To implement this in OpenMP using a strategy based on `#pragma omp for` we will resort to the `reduction` clause as it is more efficient than using `atomic` or `critical`.

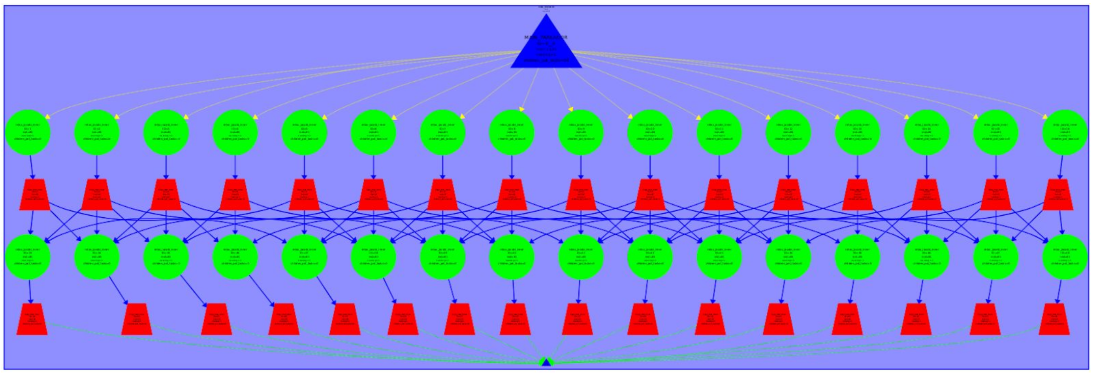


Figure 5. Tareador task decomposition graph with* `sum` *protected

Gauss-Seidel

For the Gauss-Seidel solver, similarly to Jacobi, we obtained the task dependence graph found below in Figure 6 for the coarser-grain task decomposition, that is creating only one task for every call to `relax_gauss` from `heat-tareador.c`. In this case we don't need to call `copy_mat`, and create a task for it, because we are working directly with `u` and not saving the results temporarily to `uhelp`.

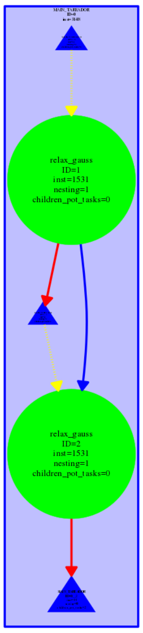
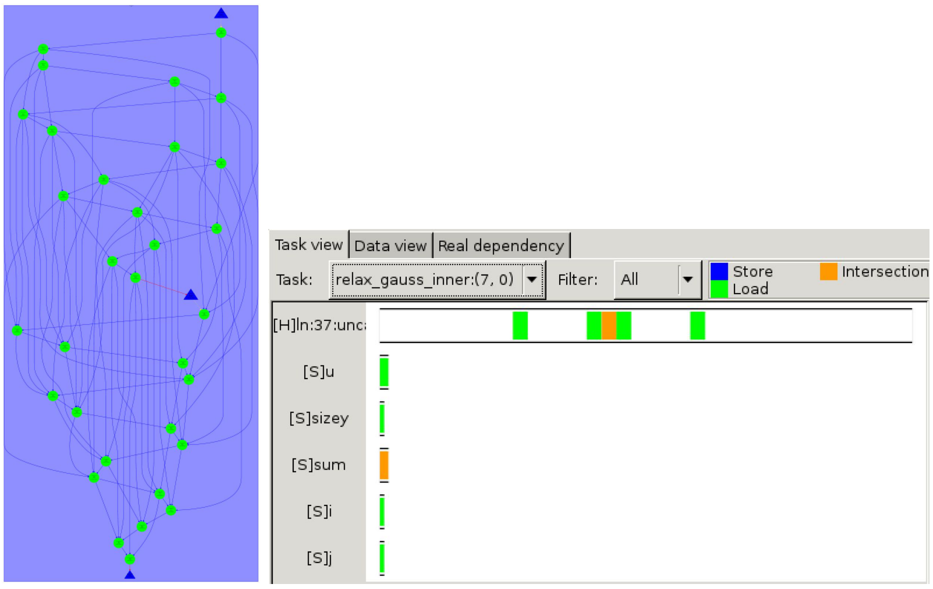


Figure 6. Tareador decomposition for the coarser-grain Gauss-Seidel solver

In like manner to what we previously did for Jacobi, we changed the code in `heat-tareador-finer.c` and `solver-tareador-finer.c` to create tasks for every iteration of the deepest for loop in `relax_gauss`, instead of creating a task for every call to `relax_gauss` in `heat-tareador-finer.c`. Executing run-tareador.sh with this changes implemented, resulted in the task decomposition that can be seen below in Figure 7.



Task: relax_gauss_inner:(7, 0)
Filter: All

Figure 7. Tareadir task decomposition for finer-grain Gauss-Seidel and Dataview for one of the nodes

As we see in the Dataview above for one of the nodes in the task decomposition graph seen beside it, the variable that is causing the serialization is `sum` as it is the only variable being written, equally to what happened with Jacobi. Thus, we will need to protect it to prevent a data race condition. To explore possible improvements in parallelization over this version, we will filter the object causing the serialization. The code that implements the finer-grain decomposition and includes the filtering of `sum` is found below in Code 2.

```c
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("relax_gauss_inner");
         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;
         tareador_enable_object(&sum);
         u[i*sizey+j]=unew;
         tareador_end_task("relax_gauss_inner");
   }}}
   return sum;
}
```

Code 2. Finer-grain Gauss-Seidel with `sum` *filtered*

The consequence of filtering `sum` is clear when looking at the decomposition, Figure 8 below, as now we see that the degree to which we can parallelize this code has significantly increased. However, it doesn't look at first sight as parallelizable as Jacobi does, even though for Jacobi we also call `copy_mat`.

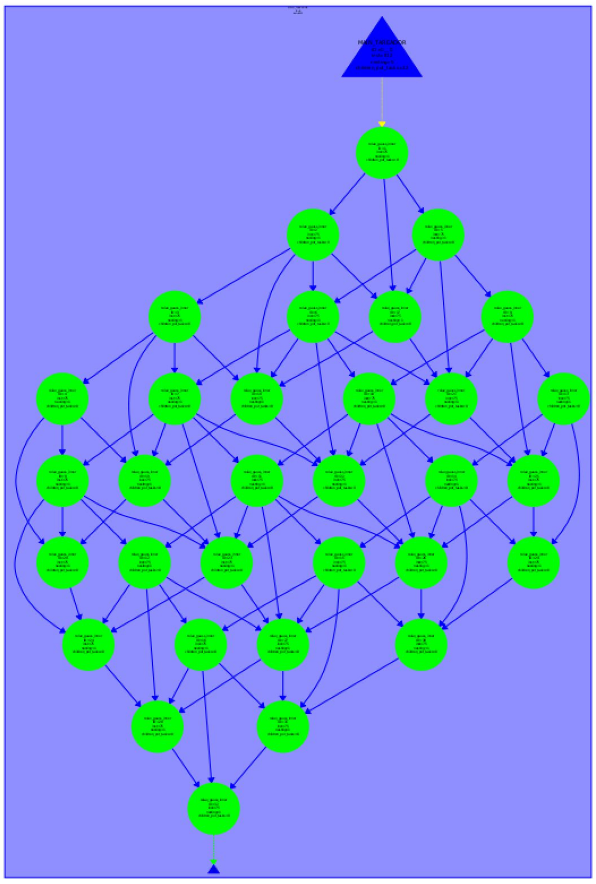


Figure 8. Tareador Gauss-Seidel finer-grain task decomposition filtering `sum`

Again, just like in the case of Jacobi, in a `#pragma omp for` strategy we would use the `reduction` clause to protect `sum` and avoid data races. If instead we were to use `#pragma omp task`, our approach would be to make `sum` private and then manually reduce all instances.

Parallelization strategies

Jacobi

In this section we make use of OpenMP constructs in order to parallelize the execution of the heat equation using *Jacobi* solver.

After understanding the macros defined in `heat.h`, it is clear that they follow a block geometric data decomposition, that for `howmany=4` looks similar to Figure 9, and it is the strategy we will follow to parallelize `relax_jacobi`. If we increase the number of threads used to `n`, then we simply divide the problem into `n` blocks, similarly to what we did in Figure 9 for `n=4`.

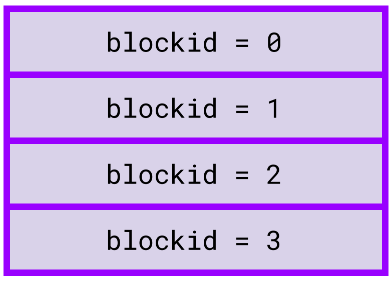


Figure 9. Block geometric data decomposition

We are ready to add the appropriate OpenMP clauses to *relax_jacobi* function and parallelize it following a block geometric data decomposition using the macros we have just observed.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey){
   double sum=0.0;
   #pragma omp parallel reduction(+:sum)
   {
     double diff;
     int howmany = omp_get_num_threads();
     int blockid = omp_get_thread_num();
     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;
}
```

Code 3: Parallelization of `relax_jacobi` ***in*** `solver-omp.c`

To parallelize it, we will start creating a parallel region at the beginning of the function, and put the declaration of the `diff` variable inside that region, to make it private for each thread. However, the `sum` variable will be kept outside to be shared between the threads and to carry the result of the function. To avoid data race problems with the `sum` variable, we use the `reduction` clause, which will create a local variables for each thread, and then add it to the global `sum` variable with an `atomic` synchronization clause.

To distribute the computation across all threads we will first modify the number of blocks created changing the value of `howmany` from 4 to the total number of threads created, which is returned by `omp_get_num_threads()`. This is especially important as it avoids the bottleneck that arises when using more than 4 threads, as otherwise only 4 of them would work on the computation. Also, we assign to the `blockid` variable from the data decomposition the id of the thread that is executing the parallel region, which we get from `omp_get_thread_num()`. This way, each thread will execute only the block that is supposed to.

After checking the correctness of the parallelization, by comparing the resulting image with the image obtained in the sequential execution, we have generated the following trace with *Paraver*.

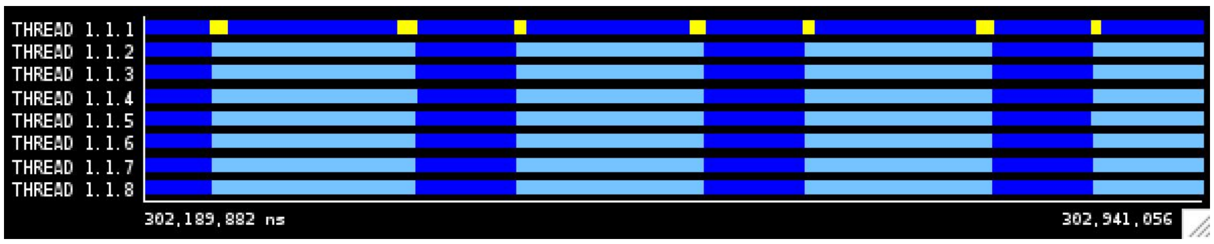


Figure 10. Paraver trace and `OMP_parallel_constructs.cfg` ***only parallelizing*** `relax_jacobi`

As we see in Figure 12, the performance of this version isn't great. That's because of the `copy_mat` function, which hasn't been parallelized and causes much of the serialization. The next step would be trying to parallelize this function. However, we realised we could rewrite `copy_mat` to make it run in constant time and not depend on the size of the matrix.

```
void copy_mat (double **u, double **v){
   double *aux = *v;
   *v = *u;
   *u = aux;
}
```
Code 4. Rewrite of `copy_mat`

Our approach to improve the performance of `copy_mat` has been to simply swap the pointers of matrices `u` and `uhelp`. We do this because we want to copy the contents of `uhelp` to `u` and, by swapping the pointers, then `u` points to the contents of `uhelp`. `uhelp` now points to the contents of `u`, but we don't really care what is now contained in `uhelp` because it will overwritten in the following iteration of `relax-jacobi`.

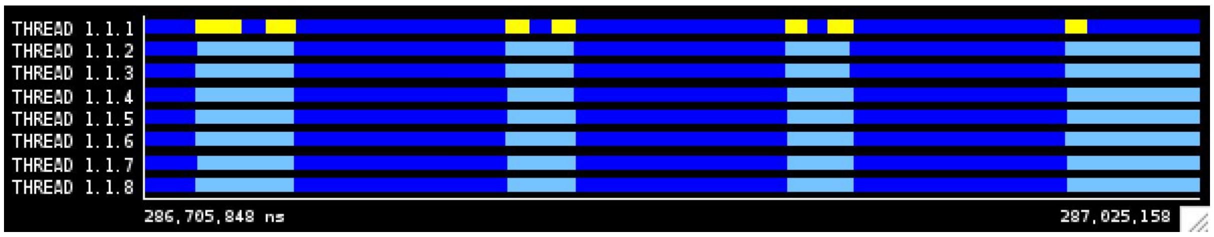


Figure 11. Paraver trace and `OMP_parallel_constructs.cfg` ***after parallelizing*** `copy_mat`

After the *Extrae* instrumented execution of our new version, we obtained the trace shown in Figure 11. As we can see, now the time the threads spend idling in between calls to relax_jacobi is way shorter than it was before, and thus the performance should be much improved.

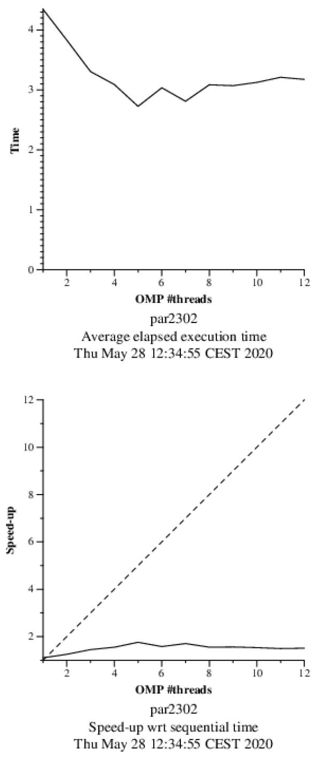


Figure 12. Strong scalability for only `relax_jacobi` ***parallelized***

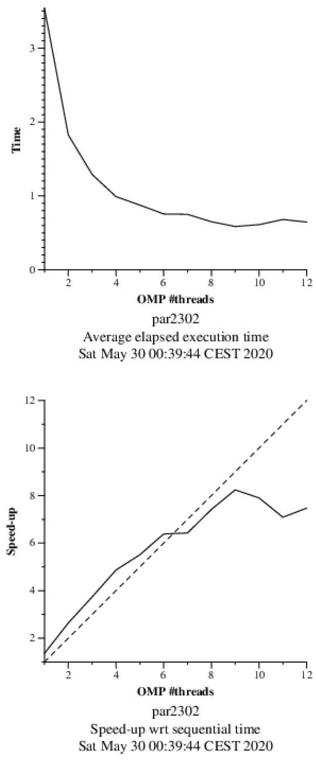


Figure 13. Strong scalability for parallelized `relax_jacobi` ***and*** `copy_mat` ***rewritten***

Finally, we'll compare the speed-up plots for both versions of our Jacobi parallelization, shown in Figures 12 and 13 above. As we see, the performance of Jacobi without parallelising `copy_mat` is very poor, the best speed-up we achieve is just short 2. Nonetheless, after rewriting `copy_mat` the improvement in performance is tremendous. The speed-up plot is super linear up to 6 threads and then it continues to increase very close to the 45 line all the way up to 9 threads. Then the speed-up decreases slightly. Hence, the improvement is clear and the performance of the Jacobi solver is much better.

Gauss-Seidel

Unlike Jacobi's algorithm, where it could be easily parallelized because it had no dependencies between iterations, in the case of the Gauss-Seidel algorithm, as we have seen in the graph of dependencies with *Tareador*, we have to take into account the dependencies that exist between iterations. The difference is that now we are modifying the value in the u array, and the next iterations depend on that value.

For Jacobi we divided the computation of the image into blocks of rows, and each thread executed it's own row-block. But in this case, this same implementation will be too slow, actually it would execute it sequentially. That's because of the dependencies that each point of the screen has with the left and above neighbours.

We will partially solve this problem dividing the image by blocks of columns as well. This way the thread that has to execute a block of rows won't have to wait until the thread executing the previous block of rows had finish with all the points. Instead, it will only need to wait for the previous thread to finish the first block of columns, so it can start running the block below it.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    #pragma omp parallel
    {
      int howmany=omp_get_num_threads();
      #pragma omp for ordered(2) private(unew,diff) reduction(+:sum)
      for (int blockid_i = 0; blockid_i < howmany; ++blockid_i) {
        for (int blockid_j = 0; blockid_j < howmany; ++blockid_j) {

          int i_start = lowerb(blockid_i, howmany, sizex);
          int i_end = upperb(blockid_i, howmany, sizex);

          int j_start = lowerb(blockid_j, howmany, sizey);
          int j_end = upperb(blockid_j, howmany, sizey);

          #pragma omp ordered depend(sink: blockid_i-1, blockid_j)
          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++) {
              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;
            }
          }
          #pragma omp ordered depend(source)
        }
      }
    }
    return sum;
}
```
Code 5. Parallelization of `relax_gauss` ***in*** `solver-omp.c`

In this first implementation we start creating a parallel region and setting the variable `howmany` to the number of threads executing the region. The value of this variable will be used both for diving the entrance by blocks of rows (and assigning one to each thread) and for dividing with blocks of columns (later we will try to improve this division).

We use the `order` clause, which allows us to specify the necessary dependencies creating an order region between the sink and the source. We create this region in the execution of each block. We indicated in the skin the dependency for the upper block. It’s not necessary to indicate the second dependency, for the left block, because the same thread is executing all the blocks from the same row in the same order. So we already honour this dependency implicitly.

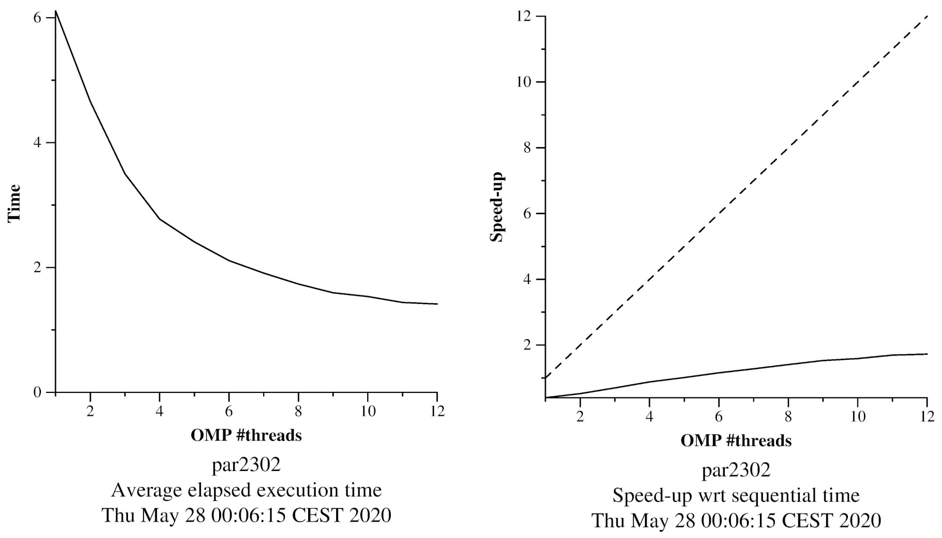


Figure 14. Strong scalability

Unfortunately, as we see in Figure 14, the scalability obtained is quite poor. We have only slightly improved the sequential execution. Of course, this is due to the dependencies of the Gauss-Seidel algorithm. The critical path would be the sequential execution of a whole column of blocks, plus the sequential execution of a whole row of blocks.

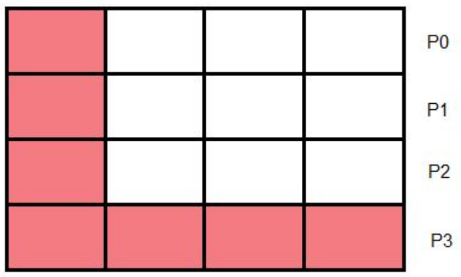


Figure 15. Representation of the critical path

Figure 15 shows a critical path (in red) of a 4-threaded execution, where both rows and columns are divided into blocks of 4. One way to reduce the critical path would be to make the blocks smaller.

So the next step will be to try to reduce the size of these blocks, taking into account the trade-off between computation and synchronization. In this case we will try to reduce the size of the column blocks, but keeping the same division of the rows (where we are dividing the entrance by the number of threads).

Below is a table with the execution of the algorithm with 8 threads, for different numbers of column blocks, that is into how many columns blocks the horizontal divisions will be split. Thus, each block will have (`sizey` / number of column division) columns.

Number of column divisions	**Execution 1 (in s)**	**Execution 2 (in s)**	**Execution 3 (in s)**	**Average (in s)**
1	6.606	6.586	6.549	6.580
8	1.719	1.652	1.670	1.680
16	1.362	1.315	1.356	1.344
24	1.253	1.252	1.247	1.250
32	1.222	1.271	1.262	1.251
40	1.130	1.207	1.128	1.155
48	1.160	1.079	1.271	1.170
56	1.143	1.396	1.176	1.238
64	1.211	1.290	1.140	1.213
128	1.554	1.619	1.571	1.581
256	2.281	2.534	2.456	2.423

Table 1. Execution time for different number of column divisions (8 threads)

We have experimented with several values, and focussing with values close to the minimum. Below, in Figures 16 and 17, we will represent this data with two plots.

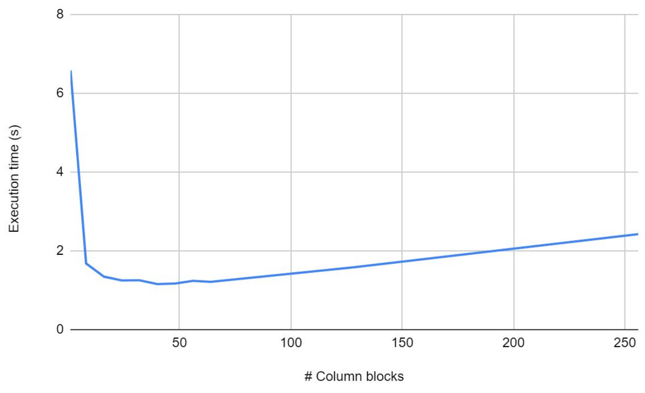


Figure 16. Full data representation

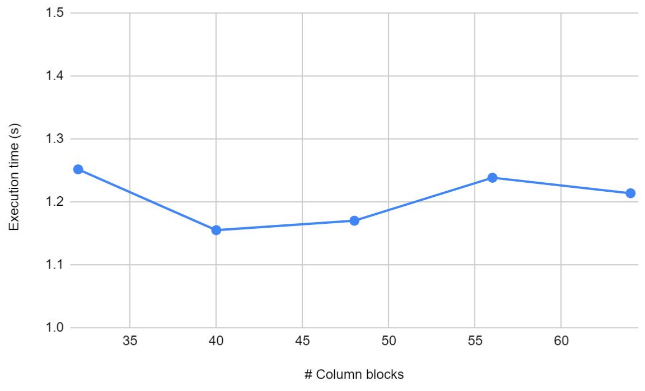


Figure 17. Zooming in on the lowest execution times

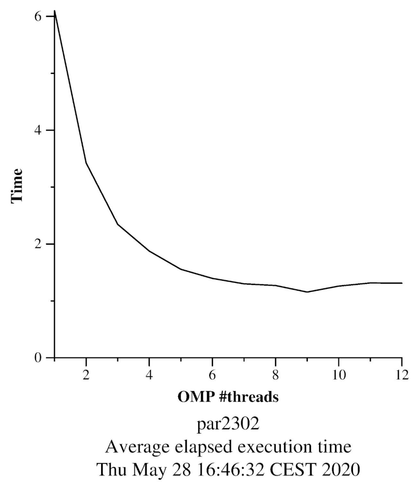
The plots shows how quickly the execution time decreases when we slightly increase the column divisions. But after exceeding the optimal value (40 blocks), the run time starts to worsen due to the synchronization overhead.

The optimal value obtained for 8 threads was 40 column divisions, meaning each row division is composed of 40 blocks. This value could be seen as 5 multiplied by the total number of threads.

```
    for (int blockid_i = 0; blockid_i < howmany; ++blockid_i) {
      for (int blockid_j = 0; blockid_j < (howmany*5); ++blockid_j) {

        int i_start = lowerb(blockid_i, howmany, sizex);
        int i_end = upperb(blockid_i, howmany, sizex);

        int j_start = lowerb(blockid_j, (howmany*5), sizey);
        int j_end = upperb(blockid_j, (howmany*5), sizey);
```
Code 6. Parallelization of `relax_gauss` ***with optimal column division***



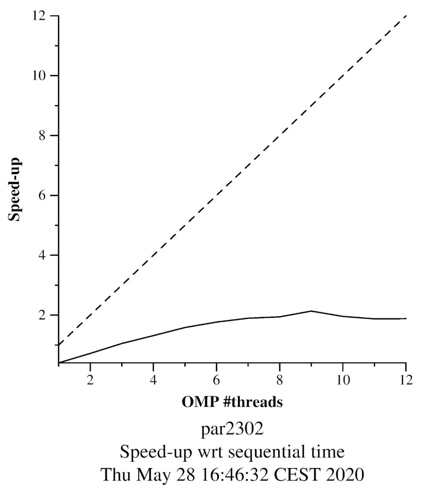


Figure 18. Strong scalability with # column blocks = `howmany` **** 5***

To check if it really improves our speed-up, we have generated the two strong scalability plots. We do see some slight improvement but it's not significant at all. We even see that when using more than 8 threads the speed-up decreases.

Performance evaluation

While Jacobi and Gauss-Seidel are different algorithms and therefore we can't really compare their execution times, what we can do is compare the speed-ups obtained when executing them with 8 threads, as it will give us an indication for the parallelization we have achieved with both of them. To do that, we will use the best performing version of each algorithm, as seen in the speed-up plots in the previous section.

Algorithm	**Execution time sequential (in s)**	**Execution time with 8 threads (in s)**	S_8
Jacobi	4.801	0.737	6.51
Gauss-Seidel with column width = `howmany * 5`	2.387	1.192	2.00

Table 2. Performance comparison for both algorithms

As we are able to observe in Table 2, the speed-up we have achieved, using 8 threads, for Jacobi is drastically better than that of Gauss-Seidel. In fact, Jacobi's speed-up more than triples Gauss-Seidel's which signals that Jacobi is way more parallelizable. That shouldn't come as a surprise, as in Gauss-Seidel we have dependencies to honour while in Jacobi we don't have any, thus introducing a certain degree of unavoidable serialization.

Conclusions

In this assignment, we have studied the parallelization of the Jacobi and Gauss-Seidel solvers for the heat equation. First, we studied their task dependency graph, which showed us that Jacobi had no dependencies while Gauss-Seidel did, specifically to the above and left neighbouring positions. With this in mind, we proceeded to attempt to parallelize both solvers.

In the case of Jacobi, we opted for a block data distribution and making use of the `parallel` construct. This provided very bad performance, which help us realise that we had some important serialization in our code: `copy_mat`. Instead of parallelizing `copy_mat`, we decided to rewrite it in order to swap the matrix pointers instead of copying the whole `uhelp` matrix into `u`. This way the complexity of `copy_mat` went from depending on the size of the matrix to being constant and thus greatly improved the performance of our Jacobi solver, even achieving super linear speed-ups for some number of threads.

For Gauss-Seidel, we opted for a block data decomposition but with the blocks broken into column blocks as well. This approach allows to honour all dependencies while speeding up the execution, as it would otherwise be sequential. The performance increase wasn't as great as we would have expected, as a matter of fact it was quite disappointing. However, we thought we could slightly improve it by empirically experimenting with different column widths. We observed that by having column blocks be 5 times that number of threads used we obtained better execution times. Nonetheless, its strong scalability was again very poor and while it showed some improvement, it was minimal. Therefore, we conclude that Gauss-Seidel is not very parallelizable, which is understandable given the many dependencies that have to be honoured.