UPC: Facultat de Informàtica de Barcelona

Lab 5 - Geometric (data) decomposition: solving the heat equation

PAR Laboratory Assignment

Oriol Moyano Núñez
Martí Ferret Vivó

ID: par2108

12/06/2018

Index

Introduction

In this document we will study the effects that data localization has on a parallel problem. That is, observing the different execution results we can get depending on the data accessed by each thread and the architecture that is being used. As a secondary focus, we will try to deepen our understanding of general parallelization principles and learn how to use different OpenMP programming directives. To those ends, first, we will make a brief explanation of some basic concepts related to data decomposition and we will analyze the problem we are trying to solve and the code that will be used to do so.

Data decomposition is an approach to parallel programming that focuses on the pieces of information each thread has responsibility of, meaning that the primary objective is reduce the execution time (same as task decomposition) but using properties like data locality or minimizing communication between threads. This family of techniques is dependent on the architecture of the machine being used, as may seem obvious because data cannot be separated from the physical hardware. This kind of parallelism has many advantages if the impact of the memory on a program is high but also brings new problems to the table as we will see through this document.

Heat diffusion based on heat equation

The program that we will parallelize computes the heat diffusion on a solid body using a two different solvers: Jacobi and Gauss-Siedel. The problem itself is not relevant just the behaviour it shows when parallelization is applied, but it is useful to know the basics on how those solvers manage to build the solution, to ease the process of modification that we will have to do.

The basic idea is, given different heat sources and other parameters like the size of the object or the temperature, the solvers try to increment the values around the heated areas, with a gradual decrease the further a point is to one of the sources. More specifically, those solvers go from the initial points (those with greater temperature) and increase the values of adjacent positions (both solvers do it differently), taking into account that the more distance is covered, the less heat there is. This process is known as relaxation, a term that will be used often in this document.

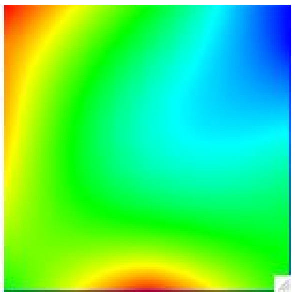


Figure 1. *Example of a result of the Jacobi solver using 2 heat sources.*

We can deduce that the code will have a substantial parallel potential due to the matrix operations (in our case we will be working with only 2 dimensions) but it is also easily seen the problems we may encounter as a result of data sharing, cache managing, load balances, etc.

Sequential heat diffusion program

Now that we have the basics down, let us take a look at the code we will be working with (the code shown below is the sequential version of it, and only the relevant parts to the parallelization tasks we will be doing):

```
while(1) {
    switch( param.algorithm ) {
          case 0: // JACOBI
              residual = relax_jacobi(param.u, param.uhelp, np, np);
              // Copy uhelp into u
              copy_mat(param.uhelp, param.u, np, np);
              break;
          case 1: // GAUSS
              residual = relax_gauss(param.u, np, np);
              break;
        }

      iter++;

      // solution good enough ?
      if (residual < 0.00005) break;

      // max. iteration reached ? (no limit with maxiter=0)
      if (param.maxiter>0 && iter>=param.maxiter) break;
  }
```

As explained before, the code does iterations relaxing the matrix to simulate heat dispersion. The most important ideas to get from this file are the following:

- The main difference between the Jacobi and Gauss solver is the need to create an auxiliary matrix in the first case (*copy_mat* function). This will be more thoroughly explained when we get to the inner functions.
- The algorithm stops when the result is good enough (residual heat is under a specific threshold) or the maximum number of iterations is reached (can be specified in the input of the program).
- Both cases are different, meaning Jacobi and Gauss, as a result, there is a parameter that controls which solver we are using. Again, this can be modified in the input file that the program will read.

The inner functions are the most interesting ones in terms of parallelization potential so let's take a look at how they work:

```
 // Function to copy one matrix into another
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++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}


 // Blocked Jacobi solver: one iteration step
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned
sizey)
{
    double diff, sum=0.0;

    int howmany=4;
    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;
}

 // Blocked Gauss-Seidel solver: one iteration step
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany=4;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
          unew= 0.25 * ( u[ i*sizey + (j-1) ]+  // left
                     u[ i*sizey     + (j+1) ]+  // right
                     u[ (i-1)*sizey + j     ]+  // top
                     u[ (i+1)*sizey + j     ]); // bottom
          diff = unew - u[i*sizey+ j];
          sum += diff * diff;
          u[i*sizey+j]=unew;
        }
      }
    }

    return sum;
}
```

With this code, we can see more clearly the difference between both solvers; on one side the Jacobi one uses a temporal matrix to increase the original values while the Gauss one, updates the argument matrix with new values on each iteration. If we take a look from the data perspective, we can see the problems on dependencies the Gauss solver will probably have, so it may look like a worse solution, but we have to take into account other factors as the copying of the whole matrix only done in the Jacobi solver. We will find more differences in their behaviours and explain them on other sections of this document.

To finish with the code explanation, on the figure below we want to show the different results we get depending on the solver used:

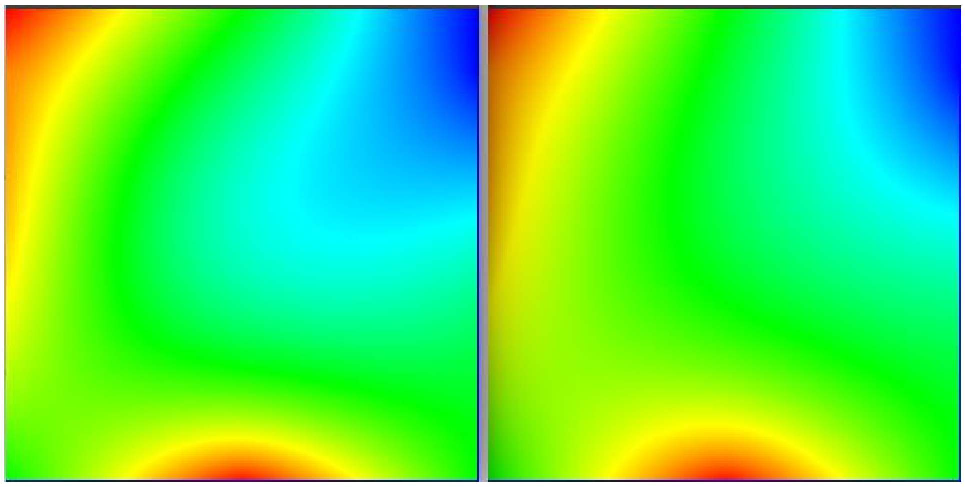


Figure 2. *Results using heat.c. On the left the Jacobi solver image, on the right, the Gauss one.*

We can see that, as a result of using the new values and adding those to the original matrix, the Gauss solver looks smoother than the Jacobi one. These two images will be further useful when the need to check the correctness of the parallel programs arises.

Parallelization analysis

Analysis of potential parallelism with Tareador

First we will analyze the potential parallelism the two proposed solvers have. In order to accomplish this, Tareador will be used (tool to study the time and load of different tasks before the program is truly parallel). These section will be divided in two different parts, one for each parallelization solver proposed above.

Jacobi

To generate the task dependency graph, we call to the tareador API with "`tareador_start_task`" and "`tareador_end_task`" to start and end a task. After that we disable the object sum to generate the task dependency graph without consider the variable sum. The relevant code to allow Tareador to simulate the parallelism is shown below:

```
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_i_j");
             tareador_disable_object(&sum);
             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;
             tareador_enable_object(&sum);
             tareador_end_task("relax_jacobi_inner_i_j");
            }
      }
    }

    return sum;
}
```

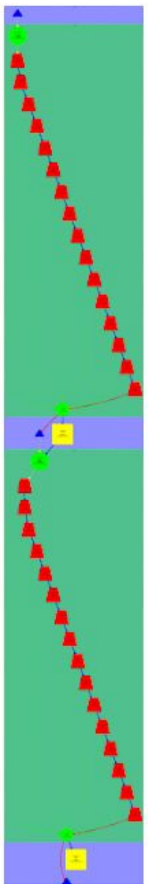


Figure 3. *Task dependency graph of the Jacobi solver (**considering** sum variable).*

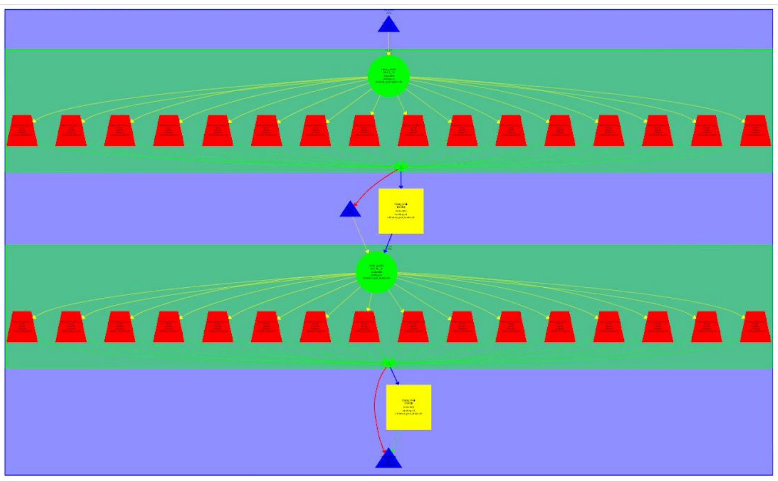


Figure 4. *Task dependency graph of the Jacobi solver (**without considering** sum variable).*

The images above, are showing the difference task dependency graph of the Jacobi solver. As we can observe in **Figure 3** (considering sum variable) there exists a huge dependence on the task almost it can not be parallelized. However, in the **Figure 4** (when we ignore the variable sum) we can see that the task dependency graph is more parallelizable. So, knowing this, we will have to consider this fact in order to achieve the maximum parallelism possible.

Gauss-Seidel

As we did in the Jacobi solver to generate the task dependency graph with Tareador, here we first consider the variable sum and then we ignore it. In the gauss-Seidel solver, we see that the variable u generates a dependency, so we disable it in the second graph like the variable sum. After that we compare the graph results. The relevant code to allow Tareador to simulate the parallelism is shown below:

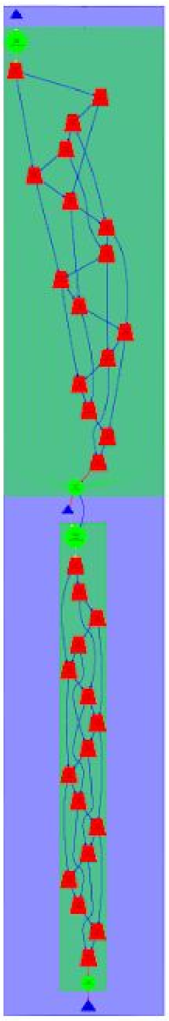
```
/*
 * Blocked Gauss-Seidel solver: one iteration step
 */
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    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_i_j");
            tareador_disable_object(&u);
            tareador_disable_object(&sum);

          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;

          tareador_enable_object(&sum);
          tareador_enable_object(&u);
          tareador_end_task("relax_gauss_inner_i_j");
        }
      }
    }

    return sum;
}
```



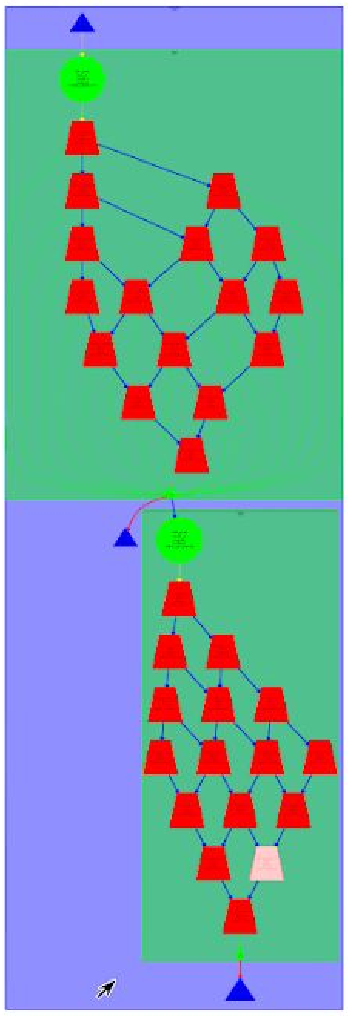


Figure 5. *Task dependency graph of the Gauss-Seidel solver (**considering** sum and u variables).*

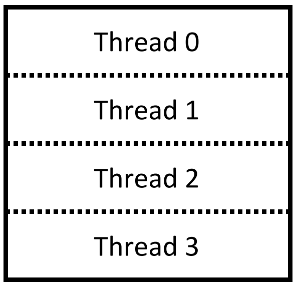
Figure 6. *Task dependency graph of the Gauss-Seidel solver (**without considering** sum and u variables).*

The images above, are showing the difference task dependency graph of the Gauss-Seidel solver. As we can observe in **Figure 5** (considering sum and u variables) there exists a dependence on the task almost. However, in the **Figure 6** (when we ignore the variables sum and u) there exists a dependence but we can see that the task dependency is more parallelizable. So, knowing this, we will have to consider this fact in order to achieve the maximum potential of parallelism.

Parallelization of Jacobi with OpenMP parallel

Data Decomposition Strategy

The data decomposition strategy that is applied to solve the problem is a geometric block data decomposition, where each block is assigned to each consecutive thread and every block is composed by the numbers of rows in the matrix divided by threads rows. So thread 0 will execute the first block of rows, thread 1 the next block and so on. Here below there is a scheme which represents the geometric block data decomposition of the matrix which a `number_of_threads =` 4.



Relevant portions of the parallel code

As mentioned before we applied a geometric block data decomposition to divide work between all threads. To achieve that, we calculate for each thread his thread id, the lowerbound and the upperbound of the for and how many elements are executed by this thread. The relevant code to achieve this strategy is shown below:

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

Execution of Jacobi solver

We have execute our version of the jacobi solver with parallelization implemented to ensure the execution and the code is correct. But, analyzing this with Paraver, we note that there is a lot of sequential part of the execution. Here below you can see the Paraver trace and the huge serial part of the thread 0:

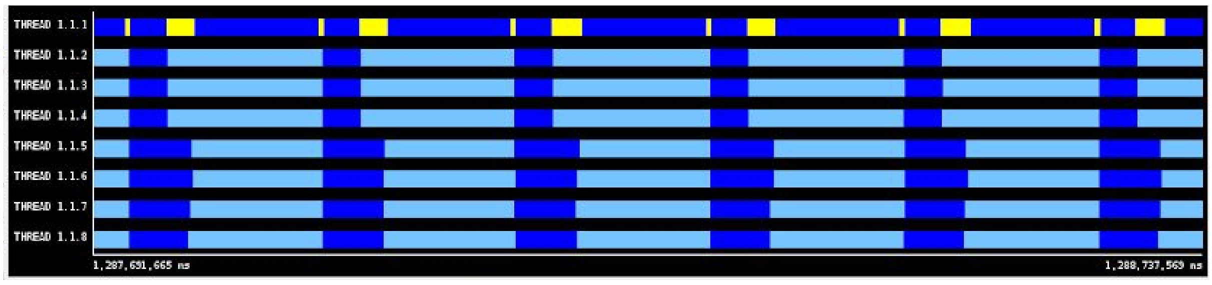


Figure 7. *Trace generated, note the huge serial part of thread 0. There are unbalance of work.*

Optimization of Jacobi solver

Relevant part of the code

The main reason of the problem that we had before is that in the first part of the program execution there is a function that takes a lot and it is not working in parallel, that function is copy_mat (function that copy one matrix into another). So to solve it we parallelize this function. The relevant code to parallelize this function is shown below:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
      #pragma omp parallel
    {
    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++) {
           v[i*sizey+j] = u[i*sizey+j];
         }
     }
    }
}
```

Note that we follow the same data decomposition strategy as we did before, each thread copies a consecutive rows of the matrix.

Analysis of the optimization

Now that we have the function copy_mat working in parallel, we proceed to execute and analyze the execution. We got these results with the analysis of the Paraver:

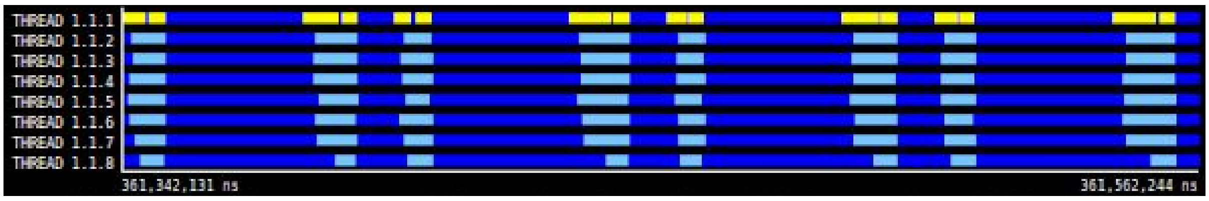
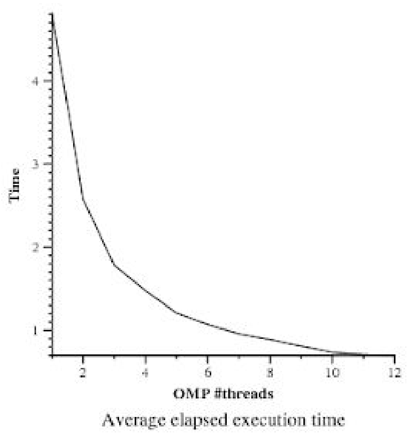


Figure 8. *Trace generated, now we fixed the unbalance generated by copy_mat.*

As you can see in **Figure 8**, now the work is more balanced and the total execution time is less.

Strong Scalability Plots

We submit the program to get the plots for strong scalability and we got these results:



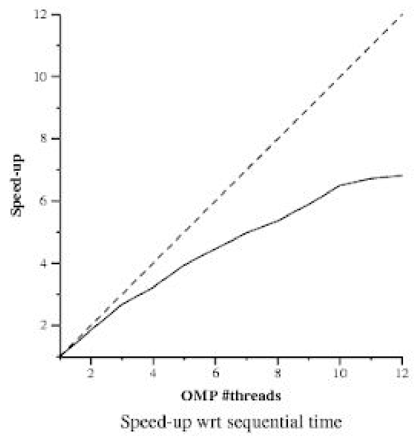


Figure 9. *Results of the submit-strong-omp.sh. Strong scalability plots of the Jacobi solver with optimization.*

In these plots we can see that the execution time of the Jacobi implementation decreases as more threads are execution the program. As consequence, the speed-up increases as more threads are. However, when the tenth thread is reached, the speed-up do not grow due to synchronization problems.

Parallelization of Gauss-Siedel with OpenMP ordered

Relevant portions of the parallel code

As we did before we applied a geometric block data decomposition to divide work between all threads. To achieve that, we calculate for each thread the lowerbound and the upperbound of the for and how many elements are executed by this thread. However, in the Gauss-Siedel strategy, we need to privatizate the variables diff and unew and make a reduction to variable sum in order to keep the execution correct. Moreover we need to add an ordered and a dependence between the tasks. The relevant code to achieve this strategy is shown below:

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

       #pragma omp parallel
       {
       int howmany=omp_get_num_threads();
       int howmanyC = 4;
       #pragma omp for ordered(2) private(diff, unew) reduction(+: sum)
       for (int blocki = 0; blocki < howmany; ++blocki) {
         for (int blockj = 0; blockj < howmanyC; ++blockj) {
              int i_start = lowerb(blocki, howmany, sizex);
              int i_end = upperb(blocki, howmany, sizex);
              int j_start = lowerb(blockj, howmanyC, sizey);
              int j_end = upperb(blockj, howmanyC, sizey);

              #pragma omp ordered depend(sink: blocki - 1, blockj) depend(sink:
               blocki, blockj - 1)
              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;
}
```

Strong Scalability Plots

We submit the program to get the plots for strong scalability and we got these results:

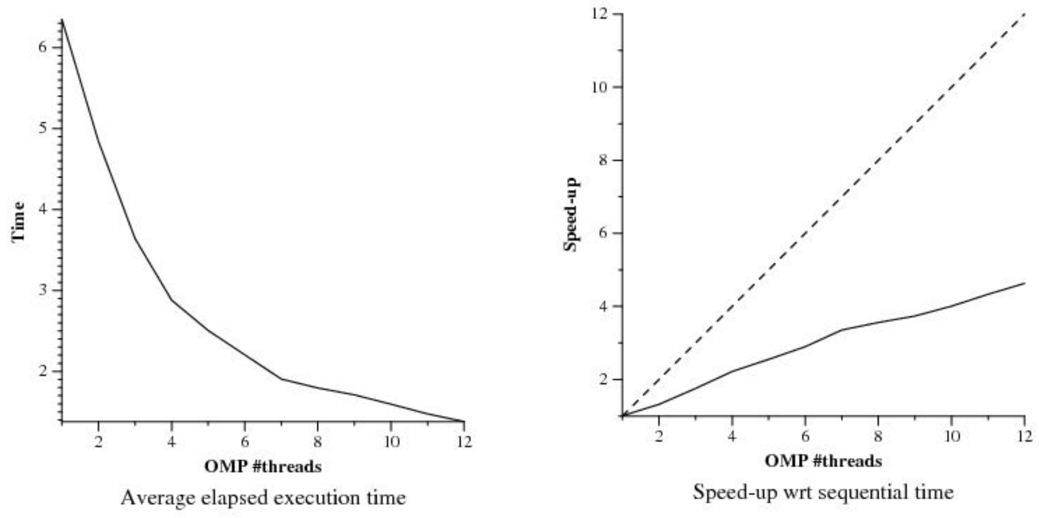


Figure 10. *Results of the submit-strong-omp.sh. Strong scalability plots of the Gauss-Siedel with optimal values.*

In these plots we can see that the execution time of the Gauss-Siedel implementation decreases as more threads are execution the program. As consequence, the speed-up increases as more threads are. If we compare this plot with the previous one (Jacobi), we can see that this implementation is less scalable, that is because there exists more dependencies between elements of the matrix.

Optimal values

To know the optimal values for the ratio of computation/synchronization we executed the program with 2500 iteration and resolution of 254 for different number of blocks and complete these table, we got these results:

#Blocks	1	2	**4**	8	16	32
Time (s)	1.438	1.287	**1.059**	1.096	1.232	1.732

So seeing the table we conclude that the best value for number of blocks is 4 or 8. That is because with a few blocks, there is a lot of computation per thread but we do not take advantage of cache locality, however, with a lot of blocks the time wasted on synchronization is bigger.

Conclusion

In this Laboratory Assignment, we have learnt about different data decomposition strategies using the heat diffusion program with Jacobi solver and Gauss-Siedel solver. We have analyzed, among other things, the scalability, performance and differences between them.

One one hand, with the Jacobi solver, we have learned how to implement geometric block data decomposition and the best way to achieve maximum performance about that, parallelizing the sequential parts of the program and getting threads balanced.

One the other hand, implementing Gauss-Siedel solver parallelized, we also have learned that the processor architecture can affect the performance of a program, so it is important to optimize values and parameters of a program depending on which architecture will run.

Bibliography

The links below have been consulted to shape this document and provide more information about the topics discussed:

- E. Ayguadé, J. Corbalán, J. Morillo, J. Tubella and G. Utrera (2017). *Geometric (data) decomposition: solving the heat equation*. Spain: UPC. [Date of reference: 09/06/2018]. Available on: <http://atenea.upc.edu/pluginfile.php/2247200/mod_resource/content/10/lab5-PAR.pdf>

- Wikipedia (2018). *Heat equations*. Estados Unidos: Wikipedia. [Date of reference: 08/06/2018]. Available on: <https://en.wikipedia.org/wiki/Heat_equation>

- Wikipedia (2018). *Gauss-Seidel method*. Estados Unidos: Wikipedia. [Date of reference: 08/06/2018]. Available on: <https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method>

- Wikipedia (2018). *Jacobi method*. Estados Unidos: Wikipedia. [Date of reference: 08/06/2018]. Available on: <https://en.wikipedia.org/wiki/Jacobi_method>

- Intel (2015). *OpenMP user and reference guide*. Estados Unidos: Intel. [Date of reference: 09/06/2018]. Available on: <https://software.intel.com/en-us/node/524542>