

CPDS

Module II: Parallelism

David Trilla
Cristóbal Ortega

January 7th, 2015
José-Ramón Herrero

Table of Contents

Initial Analysis.....	3
Analysis with Tareador.....	6
How we do it.....	6
Simulating the parallel execution.....	8
Parallelization.....	9
OMP.....	9
MPI.....	11
CUDA.....	15
Analyzing results.....	17
Conclusions.....	20
Future work.....	20

Initial Analysis

In this assignment we are going to parallelize 3 different algorithms to solve the heat diffusion problem. In a heat diffusion is treated as a matrix called `map`, where each element of the matrix represents the amount of heat on the (i,j) position. The heat diffusion needs to calculate the value of temperature in the actual position depending on its neighbours:

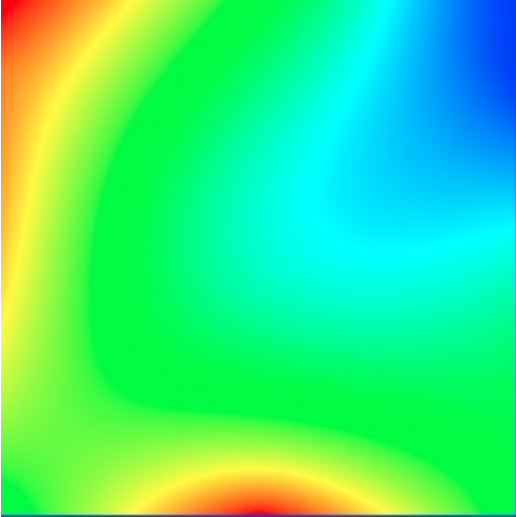
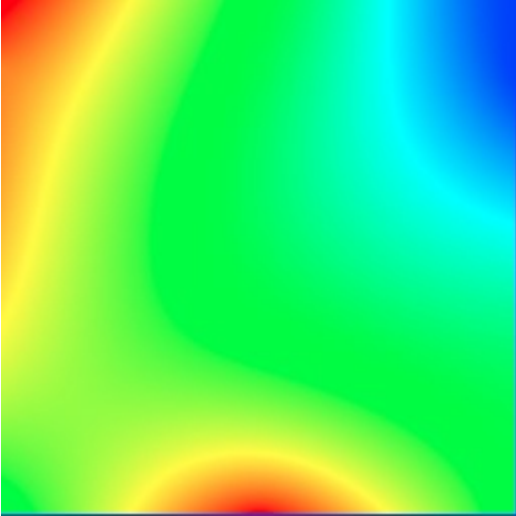
$$\text{map}[i][j] = \text{map}[i-1][j] + \text{map}[i+1][j] + \text{map}[i][j-1] + \text{map}[i][j+1]$$

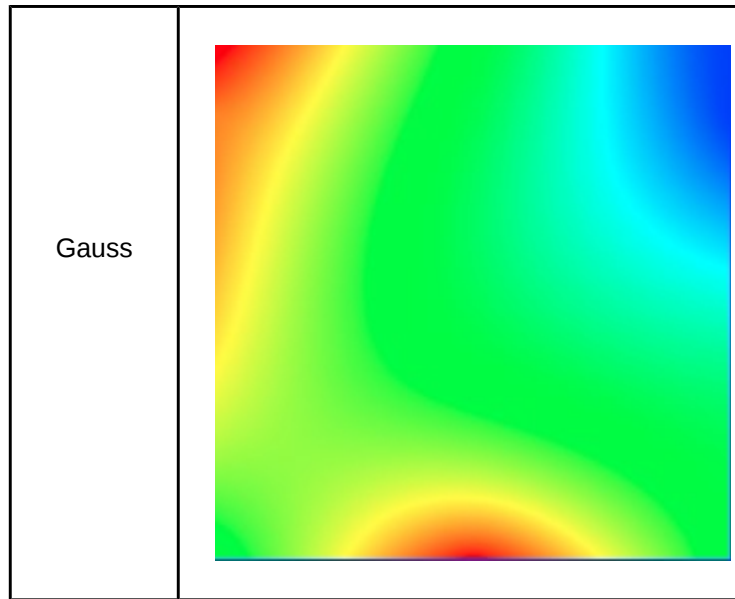
As we can see, we have some dependencies between iterations, we need the value `map[i-1][j]` to be calculated before we can do `map[i][j]`. In each algorithm that we must try to avoid dependencies with different techniques.

The particularities for every algorithm that can give us some advantages or disadvantages are the following ones:

- *Jacobi*: Here we use a new matrix to store the next iteration, so we remove the dependencies between steps. In this case we do not have dependencies between steps, but, we use a lot of memory: another matrix have to be allocated, and between steps we have to do copies of the matrixes.
- *Red-Black*: In this algorithm the dependencies are resolved as it follows. We separate “blocks” of the matrix in 2 kinds (red and black) and do first the red ones, and after red blocks are done, we do the black ones, in this way we can access to the original `map[i-1][j]`. Here we do not need 2 matrixes, but we create this dependency: We have to make sure that red blocks are finished before black blocks.
- *Gauss*: Finally, in Gauss algorithm we iterate as it was the jacobi algorithm, but we do not create an auxiliary matrix to store the new values, for this reason we need to be completely sure that `map[i-1][j]` has been calculated.

We do a first run of the 3 algorithms to see how the heat map looks:

Algorithm	Result
Jacobi	
Red-Black	



Analysis with Tareador

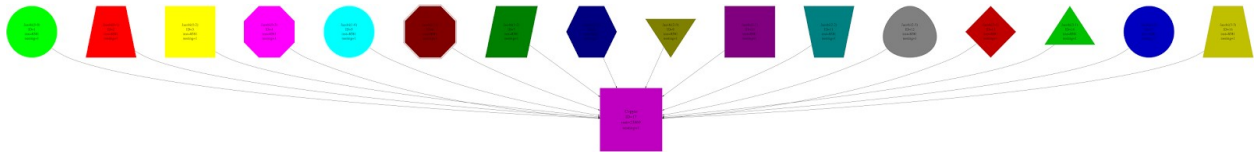
The first analysis will try to find those points in the code, that prevent parallelism to be applied, (i.e.) shared variables or dependences. By using Tareador, disabling those dependences and establishing the granularity of the tasks, we can generate various graphs indicating the weight of the tasks and the dependences between them.

How we do it

We studied the code to see possible dependencies, then we use Tareador to artificially remove them:

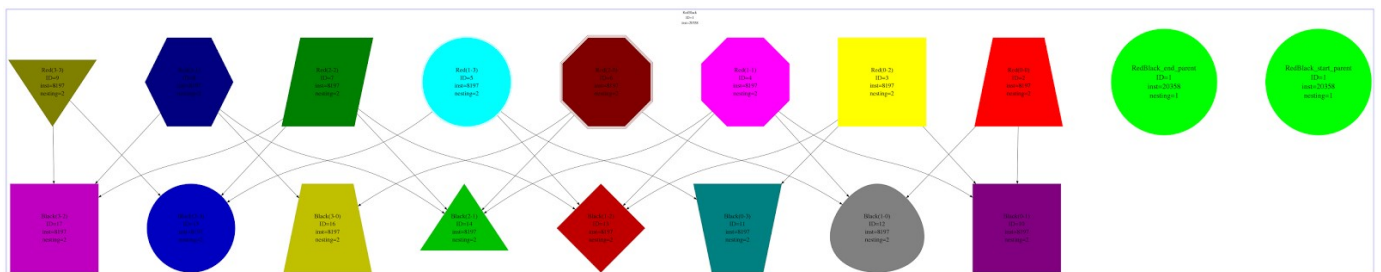
```
tareador_disable_object(&sum)
    //Possible memory access which is creating a dependency
tareador_enable_object(&sum)
```

JACOBI:



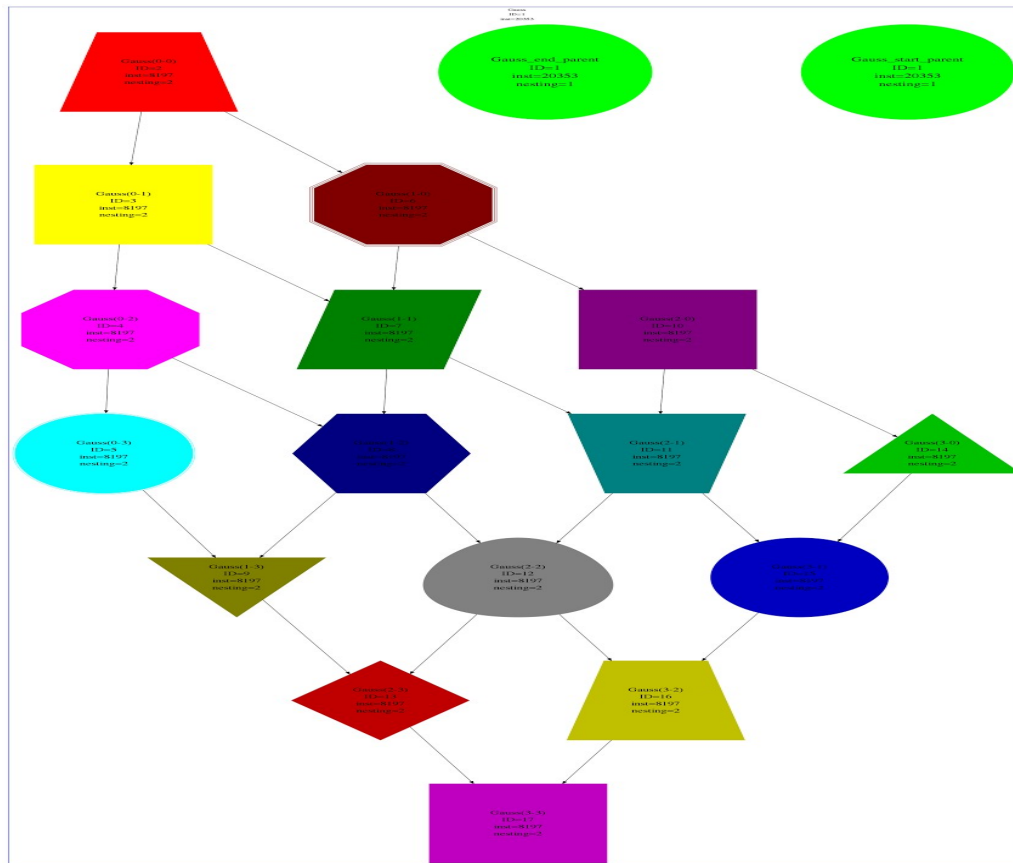
As we can see, in the Jacobi algorithm, after disabling the memory position of the variable sum, we get almost a full independent task algorithm

RED-BLACK



In Red-Black , the algorithm treats the matrix as a chessboard, where red tiles are treated first, and then the black ones, the dependences as we can see in the graph. are between the different colour tiles, so in order to parallelise this , we will have to deal with that.

GAUSS



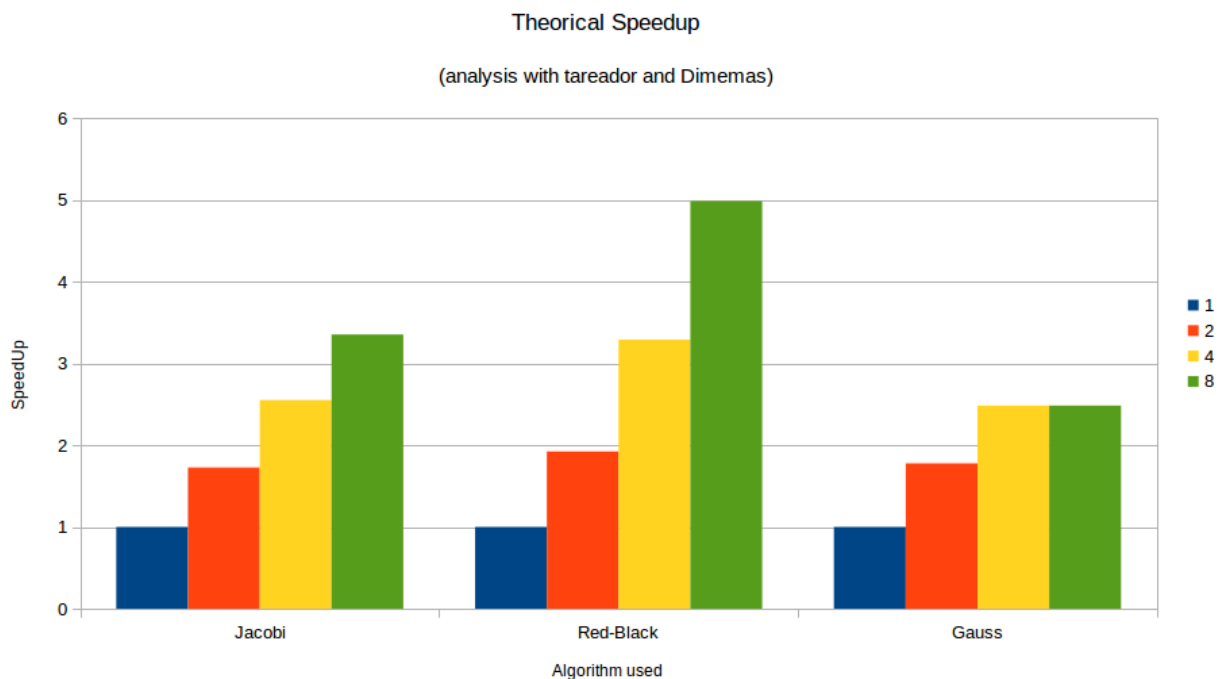
The Gauss-Siedel algorithm works with blocks to ensure that the position $\text{map}[i-1][j]$ is calculated before $\text{map}[i][j]$, so what we see is the algorithm being calculated by diagonals: first step: 0,0 second step: 0,1 and 1,0 and so on.

Simulating the parallel execution

	1	2	4	8	16
Jacobi	183.406.001 ns	106.177.001 ns	71.853.001 ns	54.691.001 ns	54.535.001 ns
Red-Black	153.922.001 ns	80.059.001 ns	46.778.001 ns	30.877.001 ns	30.825.001 ns
Gauss	153.904.001 ns	86.554.001 ns	61.963.001 ns	61.963.001 ns	61.963.001 ns

In this table we see that Jacobi has more potential than any other to be parallelized (because it has no dependencies between steps, but it needs more memory).

We also see that Red-Black and Gauss algorithms stop to obtain speedup after 4 threads, and that Red-Black is faster than Gauss.

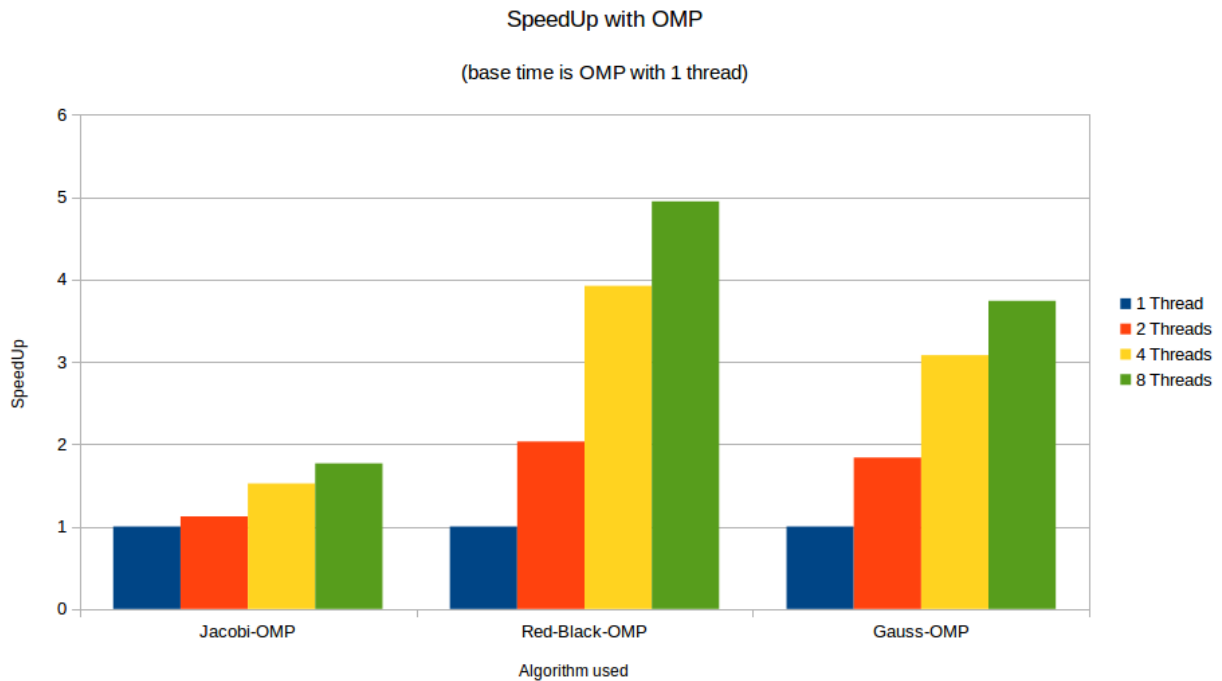


Parallelization

OMP

The results of our parallelization:

	1 Thread	2 Threads	4 Threads	8 Threads
Jacobi-OMP	6.433	5.731	4.228	3.642
Red-Black-OMP	6.787	3.341	1.731	1.372
Gauss-OMP	6.552	3.568	2.128	1.752



1.1.1. Jacobi

In the parallelization with OpenMP, we plan to guarantee these dependencies in this way:

We declare the variable `diff` as private, with this, we avoid the RAW dependency through making a local copy for each thread using the variable.

In order to solve the bottleneck made for the simultaneous access to `sum` we use the `pragma reduction(+:sum)`. It works like the clause `private` and at the end of the `pragma's`

execution, joins all the threads with the operation that we have indicated in the pragma (i.e.) + for addition to the variable sum.

1.1.2. Red-Black

For the red-black solver, there are the same two dependencies mentioned before on the Jacobi algorithm, and there are two more dependences corresponding to the variables `unew` and `lsw`. As `unew` and `lsw` establish RAW dependencies. Red-black algorithm is divided in two groups of tasks/blocks called red and black. We have to make sure that red blocks are finished before black blocks. In order to resolve the data dependencies from `unew` and `lsw` we will use the same method we used with the variable `diff` in jacobi. Again, we use the pragma reduction to the variable `sum`. In the case of data dependencies, we will use the pragma for in the two fors. Doing this, we are able to do first the red blocks than the black blocks, because at the end of each pragma for it exists an implicit pragma barrier that will force to end and synchronize the tasks assigned to the threads.

1.1.3. Gauss

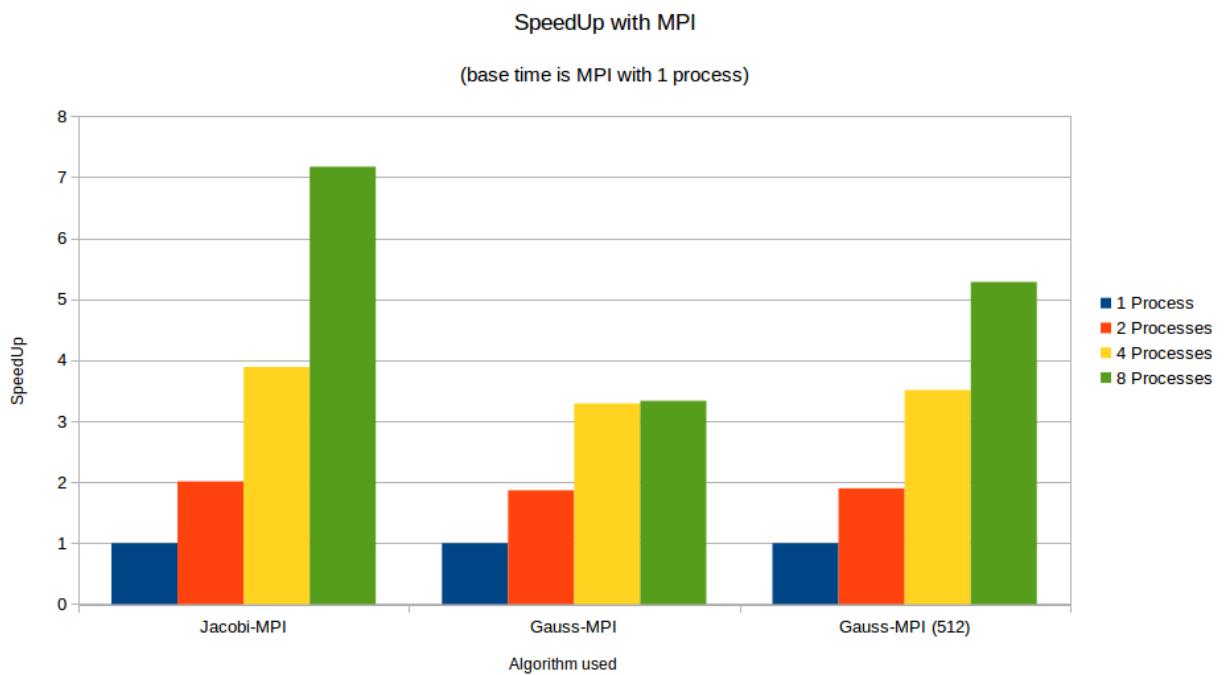
Finally, we will see about the cause of algorithm Gauss. The main obstacle of Gauss algorithm for the paralyzation of tasks is, for each data calculated $u_{i,j}$ in the matrix, we need that the data correspond to $u_{i-1,j-1}$ have been already calculated. The other data dependencies like `diff`, `unew`, `sum` are common with the Red-black and Jacobi. To be able to satisfy the mentioned dependencies, the threads must have information about the advance in the execution of the other blocks. Through the access to the vector `blocksfinished[]` a task can determinate if the dependencies have been already resolved, in the case that it did not happen the execution of that thread with dependencies stops `<while(blocksfinished[ii-1] <= jj)>` until it is able to continue. The pragma `omp flush` invalidates the cache lines not to lose the updates of the vector `blockfinished[]` done by the other threads. As a detail to finish, we add the clause `schedule(static,1)` to the pragma for. The parallel execution can go without this clause and the execution time is very similar, but, with this little change we can improve the time when we have less threads than row of blocks to deal, for example, 4 rows, and 2 threads.

MPI

The results of our parallelization:

	2 Processes	4 Processes	8 Processes
Jacobi-MPI	2.860	1.479	0.802
Gauss-MPI	3.493	1.981	1.955
Gauss-MPI (512)	28.789	15.566	10.340

We did a test with Gauss-MPI for a size of 512 to see if it really scale, because we saw that the speedup we obtained was really low, but after they test with 512 we realize that we have a lot of overhead between communications and the fact that we have to wait to the upper neighbour.



1.1.4. Jacobi

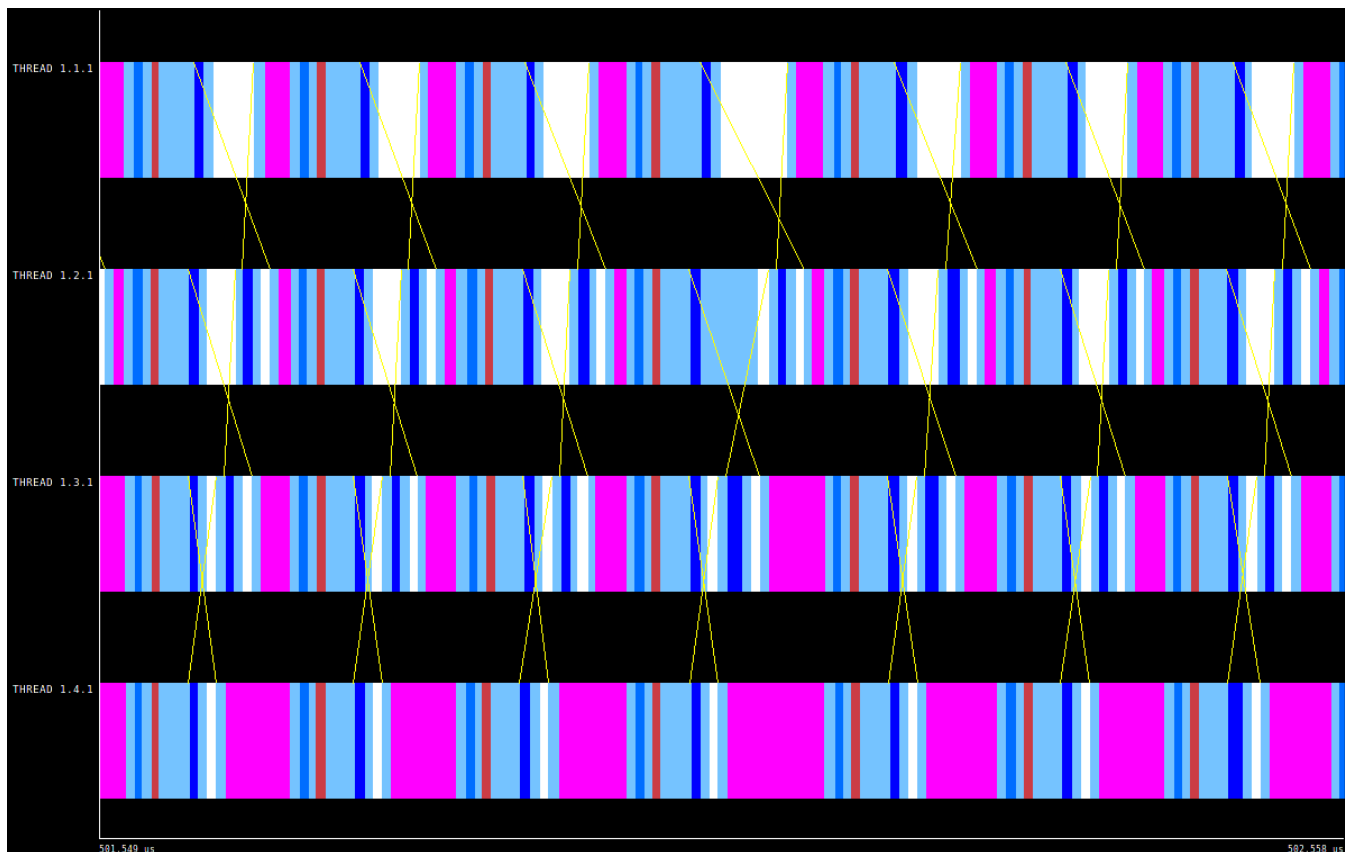
Using MPI we do not have to care about dependencies in the code because each processor will execute their own code, this means: each process executing our version of Jacobi will do it over a submatrix of the original matrix we have to compute.

So, our strategy for this case is simple, the master divides the matrix in N blocks of rows, where N will be the number of processes that the runtime of MPI called us. After knowing this, we send to each worker their space of the matrix so they can work on it.

Then, each worker and the master, will add to their part of the matrix a couple of rows to use them as borders (as in the original code), but, in this case, after computing our part we have to transfer our first and last computed rows to our upper and bottom neighbour, then they will use those rows as borders for the next iteration.

Also, we need to do a reduce for the residual value after each iteration.

When the algorithm converges we send back the data computed to the master, that recompose it and print it.



meaning:

Here, we see a trace from Extrae with 4 threads, where the colours have the next

White = MPI_Receive

Dark blue = MPI_Send

Light blue = Not MPI

Red = MPI_COMM_RANK

PINK = MPI_AllReduce

Yellow lines = communication (source to destination)

As we can observe after doing the computation (light blue) each thread has a send or a receive (and threads 2 and 3 have them both) in this step is when we send and receive the borders of our portion of matrix to do the next step of computation. After doing the work that they have, they do the Reduce for the residual and then they start again sending to their neighbours.

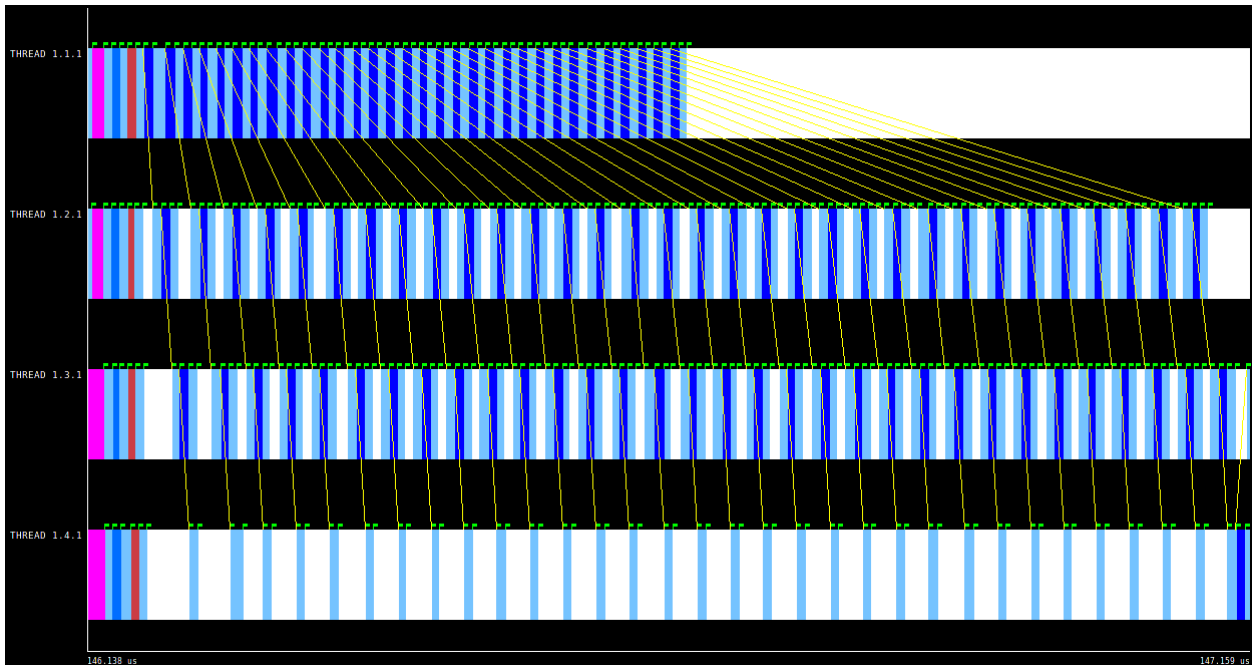
1.1.5. Gauss

Following the same scheme that in Jacobi, we do the exact steps to decompose the work to do and transfer data. But, in this algorithm we need to take in account dependencies in the solver part, so, before we start to compute we need the border updated that comes from the upper neighbour, so we do a receive if we have a upper neighbour (we are not the process 0).

And then, we do our computation in blocks of columns, after each block of columns we send to our bottom neighbour our last row computed for them to use as border.

Then, after we do a computation step in the algorithm we only send the last row of each process to their bottom neighbour to update their matrix.

As in Jacobi, we do a reduce for the residual and after converging the master asks for data to reconstruct the matrix.



Here, we see a trace from Extrae with 4 threads, where the colours have the next meaning:

White = *MPI_Receive*

Dark blue = *MPI_Send*

Light blue = *Not MPI*

Red = *MPI_COMM_RANK*

PINK = *MPI_AllReduce*

Yellow lines = *communication (source to destination)*

In order to fulfill the dependencies we see how this algorithms works when we have multiple threads: the first thread is the one who can continuously do work, and the last thread is who have to wait more.

So, when the first thread finishes a block of its work it sends to its bottom neighbour the last row computed, then the second thread can start to do some computation, and again, until the second thread does not finish the third thread cannot start.

At the end of the trace we can see a communication going from the last thread to the third thread, this is when a step in the algorithm is done we have to send the first computed row to our upper neighbour.

CUDA

The results of our parallelization:

	4 Threads/Block	8 Threads/Block	16 Threads/Block
CPU-sequential	5.685	5.685	5.685
Jacobi-CUDA	8.687	7.782	7.291
Jacobi-CUDA-Reduction	4.504	2.047	2.032

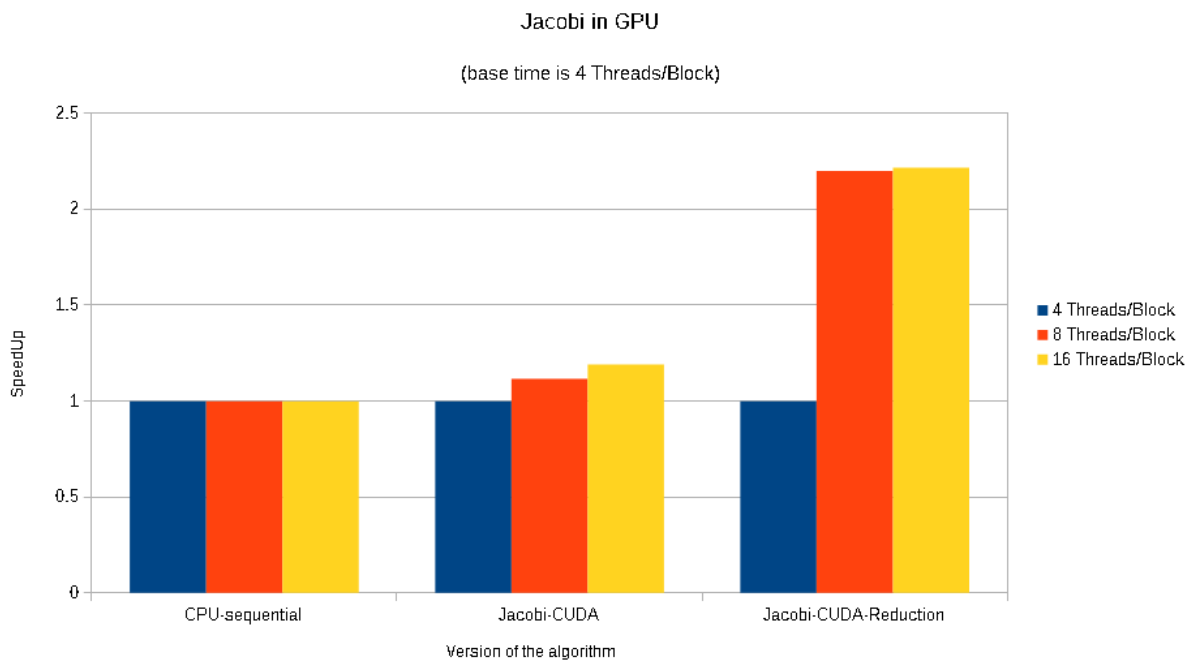
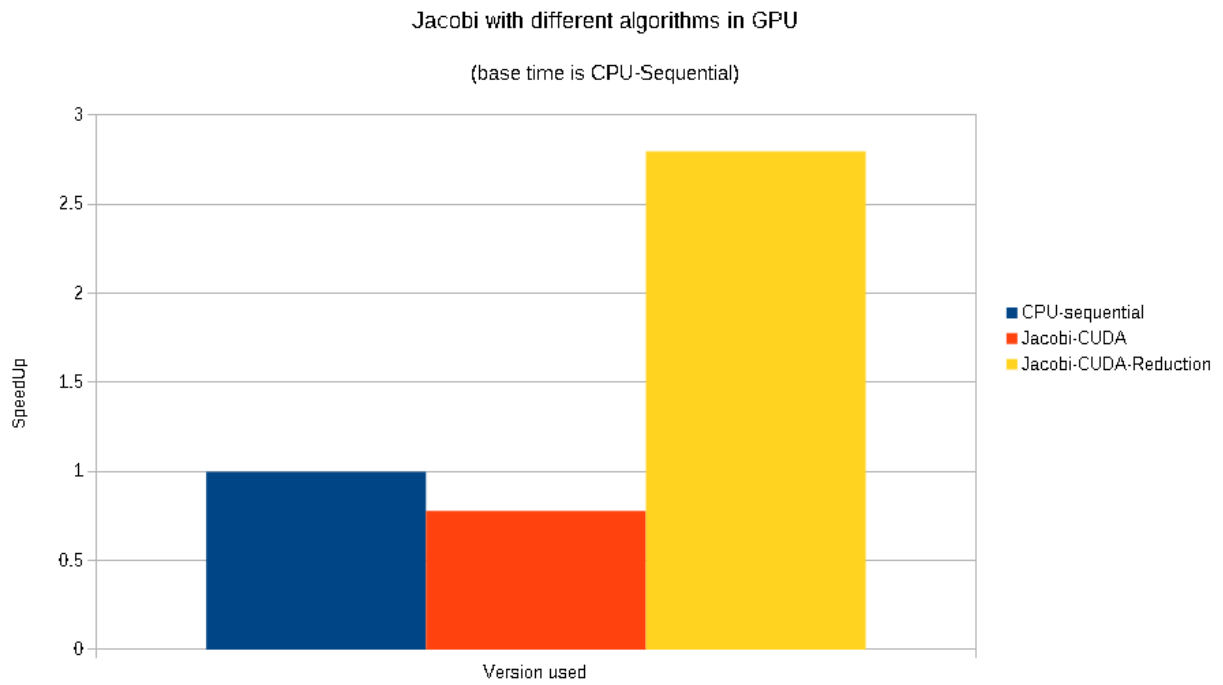
1.1.6. Jacobi

In this approach of parallelization using CUDA what we do is that each thread on the GPU computes one position of the matrix, after calling the kernel, we retrieve back the results from the Host CPU, and compute the residual value sequentially. This type of execution produces high overheads due to the need of communication between Host and Device. We have to copy back the result from the GPU each time we do an iteration.

1.1.7. Reduction

In order to avoid costly memory transfers of the whole matrix from the GPU, we will calculate the residual value for each Block on the GPU, and then make a reduction of the values of the Grid in the CPU.

This executions demands a reduction of the residual values of all the threads inside the GPU.



Analyzing results

1.2. Speedup with different numbers of processors

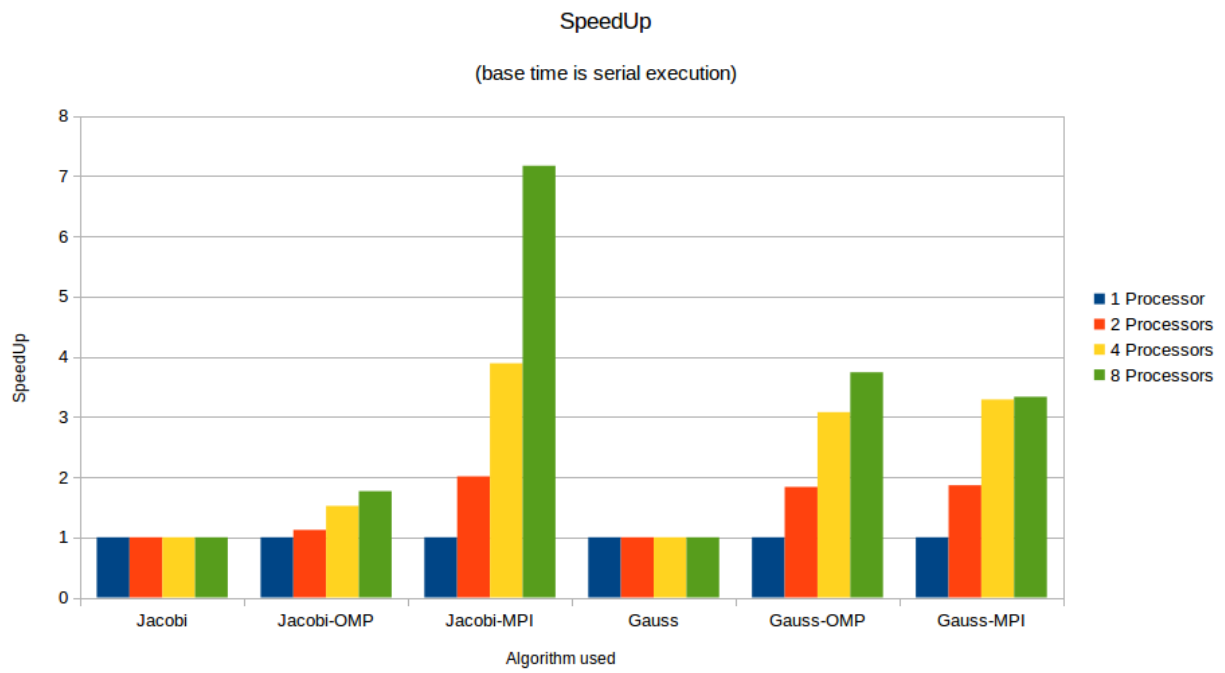
As we see in the next table, Jacobi in MPI works much better than in OMP, that may be because the reduce in MPI has less overhead than in the OMP and in MPI we are not having cache problems, because we are dividing the matrix, but in OMP everyone can write a position that will be reflected in the original matrix (main memory) and this can lead to have problems with cache misses.

And better than Gauss-MPI, this is because in Jacobi we do not have this dependency between blocks of rows, so we do not need to wait to our upper neighbour to do some computation before we can start.

Gauss-OMP and Gauss-MPI have a similar behaviour, this can be because we avoid possible problems with the cache but we are waiting data before a thread can start and we are doing the communication between processes.

	1 Processor	2 Processors	4 Processors	8 Processors
Jacobi	6.500	6.500	6.500	6.500
Jacobi-OMP	6.433	5.731	4.228	3.642
Jacobi-MPI	5.752	2.860	1.479	0.802
Gauss	6.746	6.746	6.746	6.746
Gauss-OMP	6.552	3.568	2.128	1.752
Gauss-MPI	6.516	3.493	1.981	1.955

(Processor is used to define process for MPI versions or threads for OMP versions)

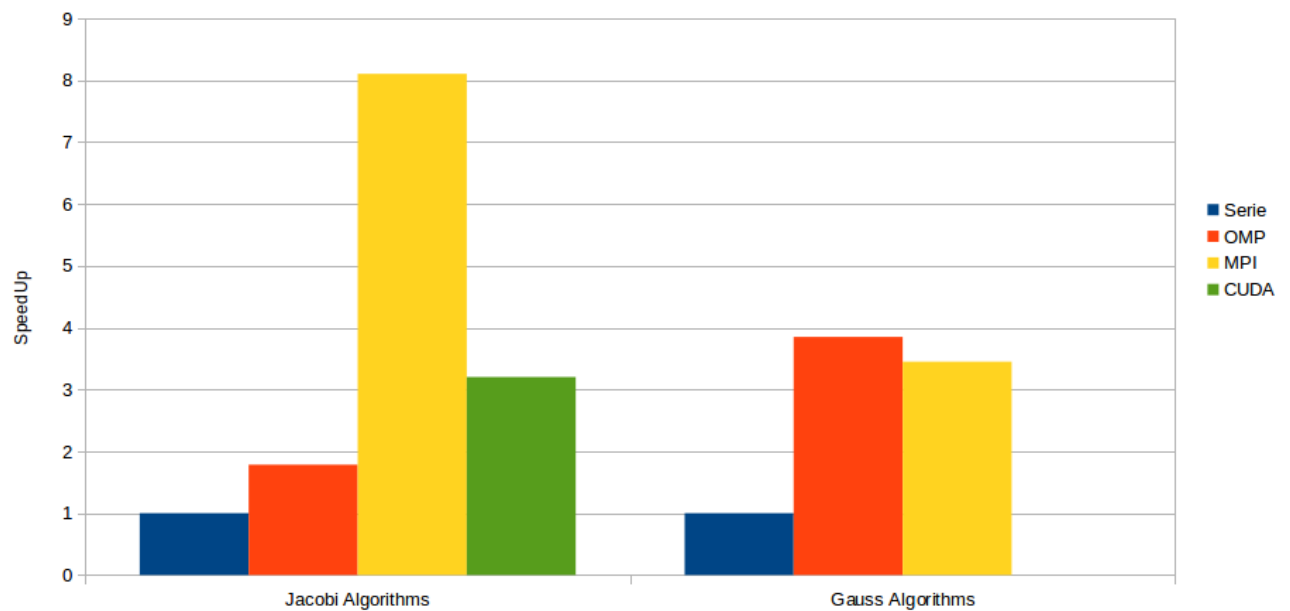


1.3. Speedup in total

	Best time (X processors or GPU)
Jacobi	6.500
Jacobi-OMP	3.642
Jacobi-MPI	0.802
Jacobi-CUDA	2.03
Gauss	6.746
Gauss-OMP	1.752
Gauss-MPI	1.955

SpeedUp with differents techniques

(base time is serial execution)



Conclusions

In this work we have seen different strategies to do parallel code with different algorithms (with and without dependencies), of course, we learnt that is easier apply those strategies without dependencies.

The strategies we applied were from divide the work in a thread level in the same processor up to use a GPU to do a lot of work at the same time, going through using different processors to do more work.

Future work

As we have seen MPI and CUDA have a lot of potential to speed up our applications, the first idea that comes to us is to apply OMP to our already done code of MPI and maybe in CUDA use different GPUs in order to parallelize even more our application.

Of course, we would like to think in how to parallelize the code of Red-Black algorithm with MPI and CUDA, our guest is that is the hardest one, we have to do more communication with MPI and in CUDA maybe more transfers between CPU and GPU.