Parallel Programming for HPC - Final report

Davide Rossi - 2023/2024

Table of Contents

1. Distributed Matrix-Matrix Multiplication

- 1.1. Introduction
- 1.2. Matrix-Matrix Multiplication using MPI
- 1.3. CPU baseline: naive algorithm
- 1.4. CPU improvement: BLAS
- 1.5. GPU version
- 1.6. Results
- 1.7. How to run

2. Jacobi's Algorithm with OpenACC

- 2.1. Introduction
- 2.2. Jacobi's algorithm
- 2.3. Distribute the domain: MPI
- 2.4. Move to GPU: OpenACC
- 2.5. Results
- 2.6. How to run

3. Jacobi's Algorithm with One-Sided MPI

- 3.1. Introduction
- 3.2. Jacobi's algorithm
- 3.3. Distribute the domain: MPI
- 3.4. Results
- 3.5. How to run

1. Distributed Matrix-Matrix Multiplication

1.1. Introduction

The first assignment consists of implementing a distributed matrix-matrix multiplication, using the MPI library to communicate between processes. More precisely, 3 versions of the algorithm are required:

- a basic version with the naive algorithm (triple loop);
- an improved CPU version using BLAS library;
- a GPU version using CUDA and CUBLAS library.

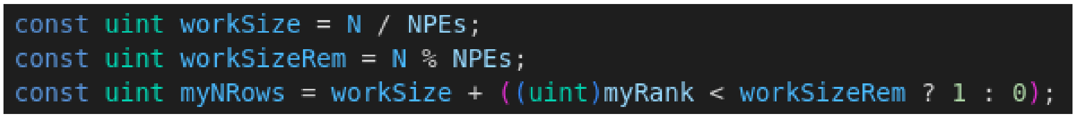
Before digging into the implementation of the three versions, let's first describe the problem and how to solve it.

1.2. Matrix-Matrix Multiplication using MPI

Matrix-matrix multiplication is a fundamental operation in linear algebra and a good exercise to implement in a distributed environment. It consists in computing $C = A \times B$, where A is a $m \times n$ matrix, B is a $n \times l$ matrix and the output C is a $m \times l$ matrix. The implementation of a distributed matrix-matrix multiplication lies on two main concepts:

- matrices are saved by rows in contiguous memory;
- each of the three matrices is distributed among the processes.

For this assignment, we will consider the matrices to be distributed *by rows* among the processes, hence each process will have a submatrix, which we will call `myA`, `myB` and `myC`, with a fixed number of rows of each matrix (equal to the number of rows of the entire matrix divided by the number of processes). Since in general the number of rows of the matrices is not divisible by the number of processes, some processes will actually have one more row than the others:



```
const uint workSize = N / NPEs;
const uint workSizeRem = N % NPEs;
const uint myNRows = workSize + ((uint)myRank < workSizeRem ? 1 : 0);
```

Figure 1: How different processes share the work

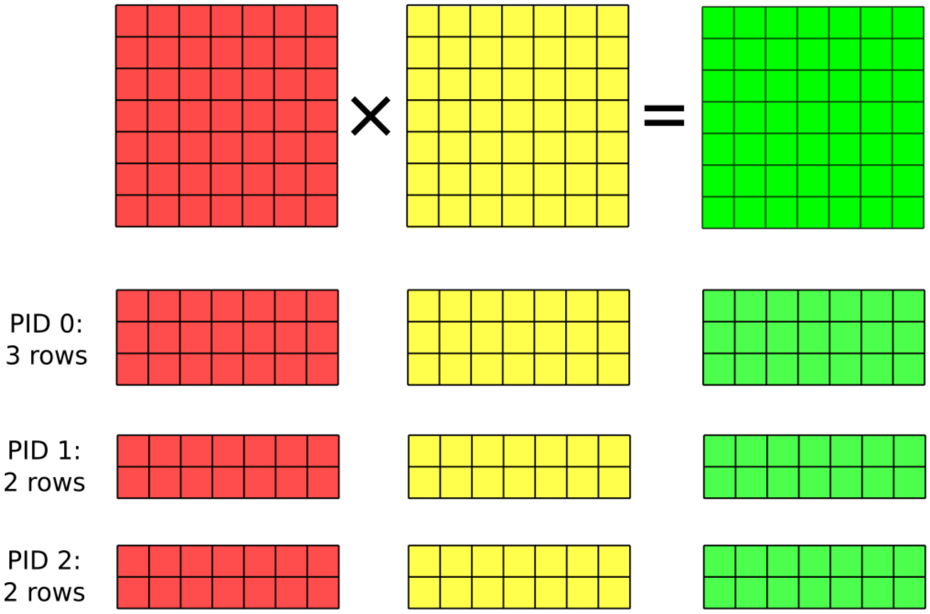


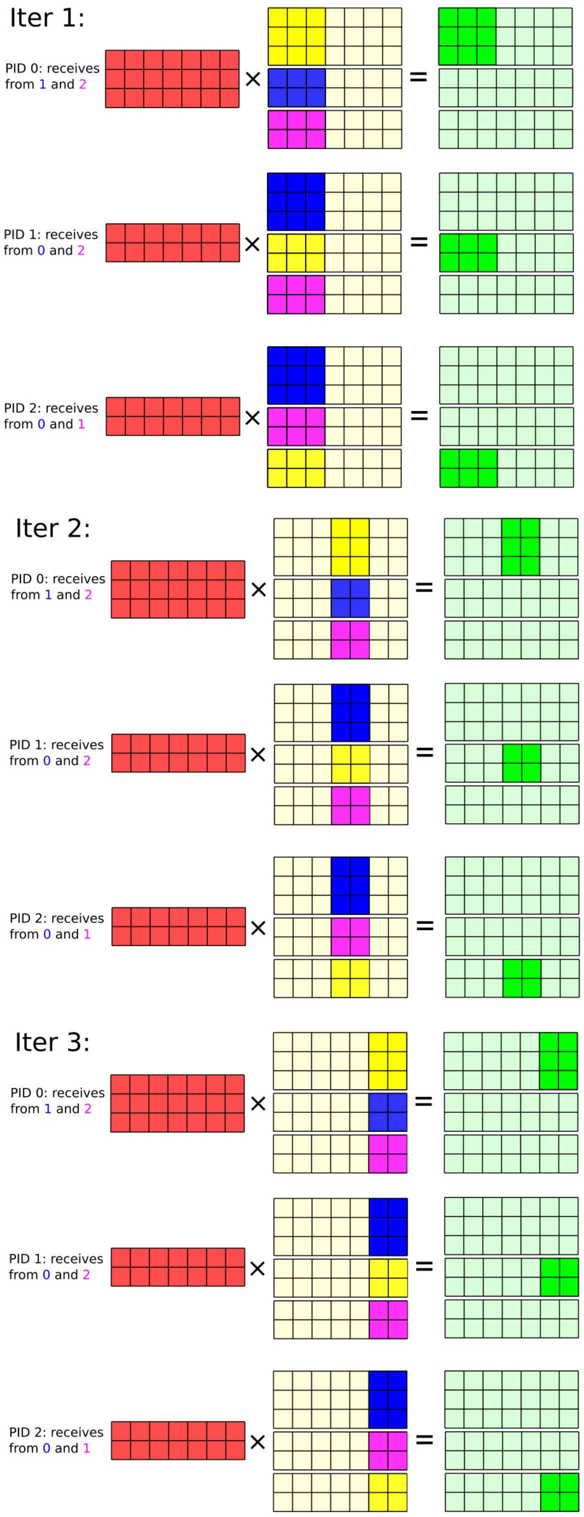
Figure 2: Distribute matrix-matrix multiplication

The idea to compute the product is the following: iterate over the number of processes: at each iteration, each process:

- re-builds a group of columns of B, named `columnB`, by gathering the necessary part from all the other processes;
- computes `myCBlock = myA × columnB`;
- places `myCBlock` in `myC`: the union of the `myCBlocks` of the current iteration will give a group of columns of the final matrix C.

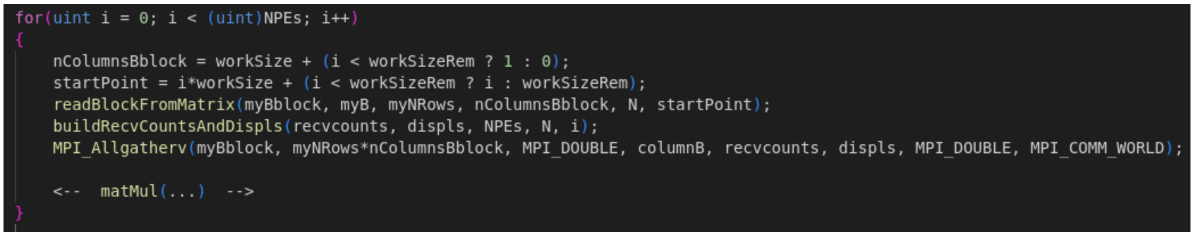
Essentially, C matrix is built by columns: at iteration `i+1`, for `i=0,...NPEs-1`, each process computes its `myNRows` rows of a block of k columns, where k is the worksize of the `i`-th process.

For example, the product in the picture above is computed in 3 iterations, as:


Iter 1:
PID 0: receives from 1 and 2
×
=
PID 1: receives from 0 and 2
×
=
PID 2: receives from 0 and 1
×
=
Iter 2:
PID 0: receives from 1 and 2
×
=
PID 1: receives from 0 and 2
×
=
PID 2: receives from 0 and 1
×
=
Iter 3:
PID 0: receives from 1 and 2
×
=
PID 1: receives from 0 and 2
×
=
PID 2: receives from 0 and 1
×
=

where `columnB` is made by the current process part, in yellow, and the parts sent by the other two processes, in blue and pink, and `myCBlock`, in green, is computed and placed in the correct position in `myC`. Note that no process will ever store any of the matrices in their entirety, but only the part they need to compute their part of the product.

The code that executes the iterations is:



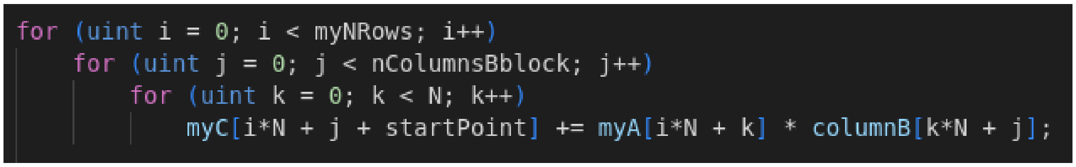
```
for(uint i = 0; i < (uint)NPEs; i++)
{
    nColumnsBblock = workSize + (i < workSizeRem ? 1 : 0);
    startPoint = i*workSize + (i < workSizeRem ? i : workSizeRem);
    readBlockFromMatrix(myBblock, myB, myNRows, nColumnsBblock, N, startPoint);
    buildRecvCountsAndDispls(recvcounts, displs, NPEs, N, i);
    MPI_Allgatherv(myBblock, myNRows*nColumnsBblock, MPI_DOUBLE, columnB, recvcounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);

    <--  matMul(...)  -->
}
```

Figure 3: Matrix-matrix multiplication base code

Where the `matMul` part branches according to the version of the algorithm we are implementing. Let's have a look at some details about the three versions.

1.3. CPU baseline: naive algorithm

The basic version of the algorithm is the naive implementation of the matrix-matrix multiplication, using the triple loop:



```
for (uint i = 0; i < myNRows; i++)
    for (uint j = 0; j < nColumnsBblock; j++)
        for (uint k = 0; k < N; k++)
            myC[i*N + j + startPoint] += myA[i*N + k] * columnB[k*N + j];
```

Figure 4: Naive product

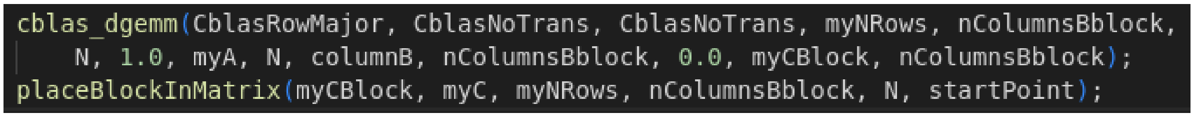
`startPoint` is a shift that allows to directly position the computed values in `myC`, without using the support matrix `myCBlock`. Except for this, the code is straightforward: each process computes its `myCBlock`, with size `myNRows x nColumnsBblock`, by iterating over the rows of `myA` and the columns of `columnB`.

1.4. CPU improvement: BLAS

The improved CPU version uses the BLAS library to compute the matrix-matrix multiplication.

The BLAS library is *"...a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products,* *linear combinations, and matrix multiplication... (from Wikipedia)"*, and constitutes the de-facto standard for linear algebra libraries.

The routine we are interested in is `dgemm`, which computes the matrix-matrix product of two matrices with double-precision elements. The code here is just a little bit more complex than the basic version: product and `myCBlock` placement are split in two different steps:



```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, myNRows, nColumnsBblock,
    N, 1.0, myA, N, columnB, nColumnsBblock, 0.0, myCBlock, nColumnsBblock);
placeBlockInMatrix(myCBlock, myC, myNRows, nColumnsBblock, N, startPoint);
```

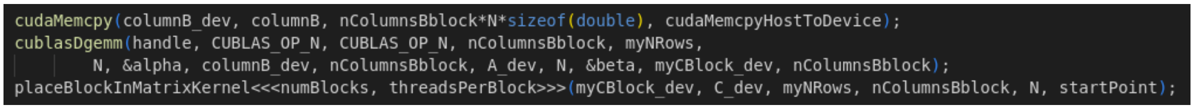
Figure 5: Product with BLAS library

We first compute the product and store it in `myCBlock`, then we place `myCBlock` in `myC` using the `startPoint` shift.

Notice that we are specifying to `dgemm` that we don't want to transpose the matrices. This is done since we want to settle in a scenario were the original matrices are already given, all in the same format (a fixed number of rows for each process), hence gathering is necessary to perform the product.

1.5. GPU version

GPU execution, which is done with CUDA and CUBLAS library, requires one more step with respect to the previous version:



```
cudaMemcpy(columnB_dev, columnB, nColumnsBblock*N*sizeof(double), cudaMemcpyHostToDevice);
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, nColumnsBblock, myNRows,
        N, &alpha, columnB_dev, nColumnsBblock, A_dev, N, &beta, myCBlock_dev, nColumnsBblock);
placeBlockInMatrixKernel<<<numBlocks, threadsPerBlock>>>(myCBlock_dev, C_dev, myNRows, nColumnsBblock, N, startPoint);
```

Figure 6: Product with CUBLAS library

We first copy `columnB` to the GPU, then we compute the product and place it in `myCBlock` as in the previous case. Some interesting points to notice are:

- all the matrices have already been preallocated on the GPU at the beginning of the execution, hence the only thing we are missing is the copy of `columnB`, which is built on the CPU at each iteration and then moved to the GPU;
- `cublasDgemm`, the CUBLAS routine that performs the product, takes as input the matrices in column-major format by default, and we don't want to transpose them to avoid losing performances, hence we perform the product in the inverse order, exploiting the fact that $C = A \times B$ is equivalent to $C^T = B^T \times A^T$: in this way the product output, which is saved in `myCBlock_dev`, is already in the correct format to be placed in `C_dev` (the GPU memory location of `myC`), but you must be careful to correctly set the leading dimensions of the matrices in the `cublasDgemm` call;

- to access myCBlock_dev and to modify C_dev we need to use a kernel function, since we are working on the GPU. Hence, we are working exclusively on the GPU for the product and the placement of myCBlock_dev in C_dev: only at the end of the program C_dev is copied back to the CPU.

1.6. Results

In this section we will analyze the results of the three versions of the algorithm. The code has been run on the Leonardo cluster, with up to 16 MPI tasks allocated one per node, for CPU versions, and up to 32 MPI tasks allocated four per node, one per GPU card, for the GPU version. The matrices have been generated randomly at each run and the execution time has been measured with the MPI_Wtime function. The tests have been done on matrices of various sizes, in order to compare CPU and GPU performances and also evaluate the scalability. The maximum time among all the MPI processes has been plotted. However, I have also collected data regarding the average time and they have showed the same behavior, meaning the workload is correctly distributed among the processes, for this reason they have not been plotted.

To easily identify the different parts of the code and plot them I have used some terms, here a brief explanation of them is given, in order of appearance in the code:

- initCuda: initialization of Cuda, with cudaGetDeviceCount and cudaSetDevice;
- init: initialization of the three matrices (done on CPU);
- initComm: initialization of myBblock, recvcounts and displs for the communication;
- resAlloc: everything related to the allocation of the matrices, both on CPU and on GPU (hence malloc, cudaMalloc, cublasCreate and cudaMemcpy);
- gather: gathering of myBblock into columnB from all the processes;
- dGemm: computation of the product, with triple loop (naive), cblas_dgemm (cpu) or cublasDgemm (gpu);
- place: placement of myCBlock inside myC.

1.6.1. CPU - Naive

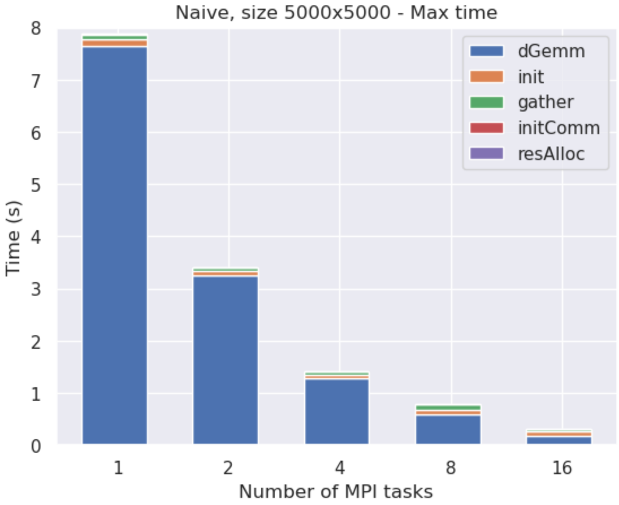


Figure 7: Naive algorithm, total execution time

As we can see, almost all the execution time is occupied by the computation of the product. Let's try to remove it to see how the other parts behave:

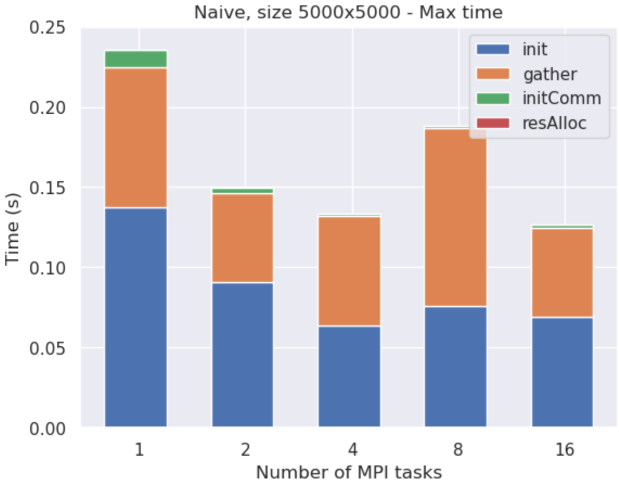


Figure 8: Naive algorithm, execution time without product

Excluding `dGemm` time, `init` and `gather` seem to be the most time consuming parts of the code (still nearly 2 orders of magnitude far from product part though). `init` is scaling a bit, but the matrix is way too small to expect something better.

In order to understand how the code scales with the size, let's also have a look at the results for 10000x10000 matrices: since the algorithm complexity is $O(N^3)$, we expect `dGemm` part to grow about 8 times:

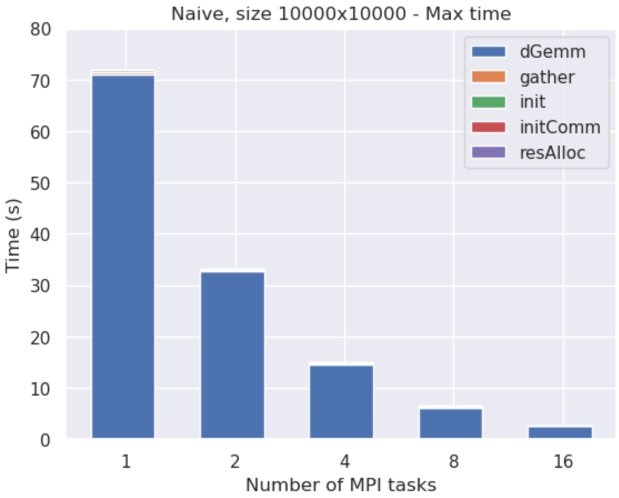


Figure 9: Naive algorithm, total execution time

Without the product part:

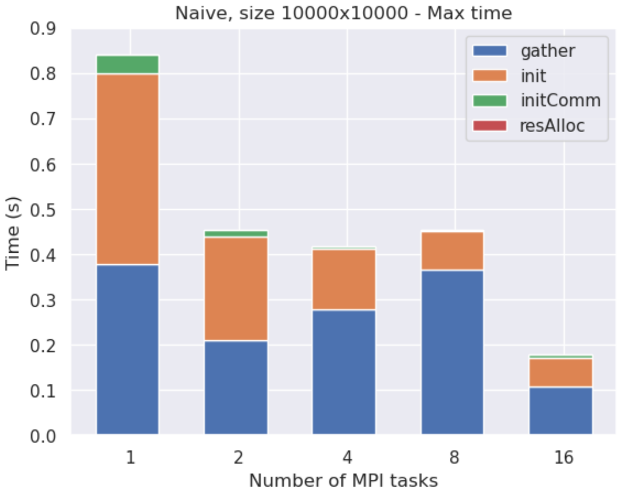


Figure 10: Naive algorithm, execution time without product

`init` seems now to scale pretty well, as expected.

1.6.2. CPU - BLAS

Let's now analyze the results for the CPU version with BLAS library:

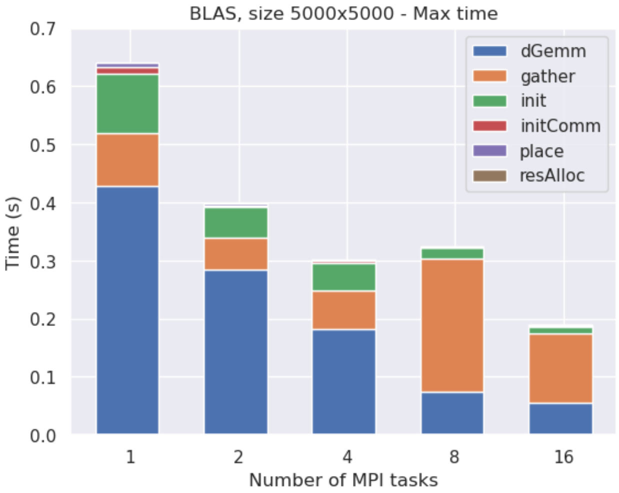


Figure 11: Algorithm with BLAS, total execution time

Also in this case, `dGemm` is the most time consuming part of the code, as we would expect. However, `dGemm` time is now ~20 times smaller than in the naive case, hence `gather` and `init` become quite significant now.

Notice that:

- `place` time was not present in the naive case, since the product was directly placed in `myC`, while in this case we first compute the product and then place it in `myC`. However, the time spent in `place` is negligible with respect to the time spent in `dGemm`, `init` and `gather`;
- for both naive and CPU version, `resAlloc` time is practically zero, since the matrices are allocated only in the CPU. We'll see that this won't be the case in the GPU version.

Let’s see how the code scales with the size:

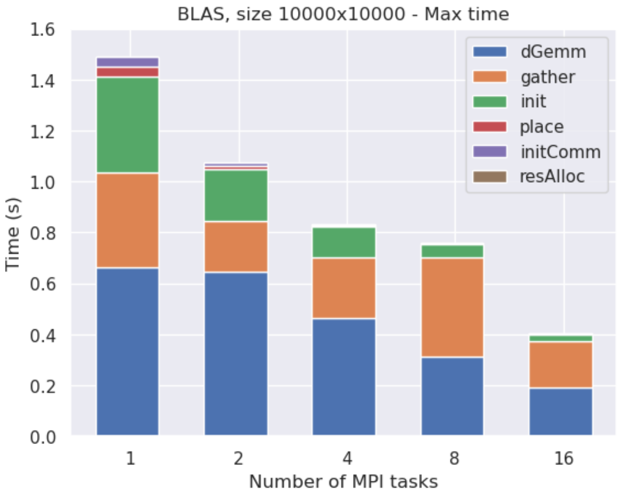
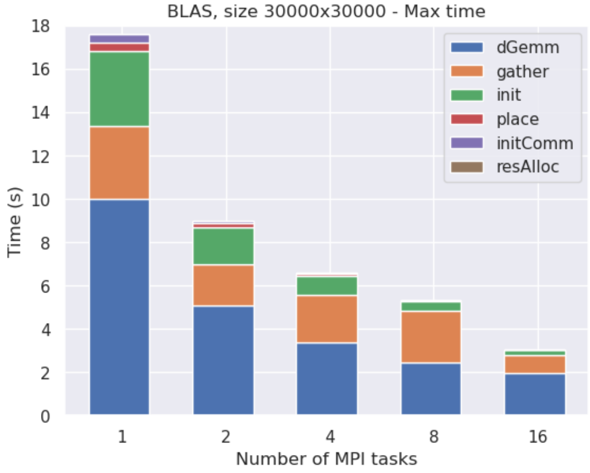
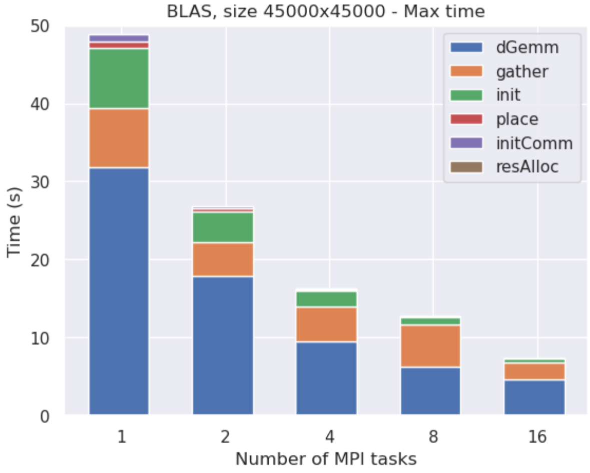


Figure 12: Algorithm with BLAS, total execution time

Results are a bit better than before but `gather` is still quite impactant. Let’s try with bigger matrices:





As we can see, with bigger matrices we are able to obtain a pretty decent scalability, although the `gather` part becomes quite impactant at large number of processes, thus becoming the real bottleneck as we would expect.

1.6.3. GPU

Finally, let's analyze the GPU version: first of all, let's see the results for the 5000x5000 matrices in order to compare them with the previous cases:

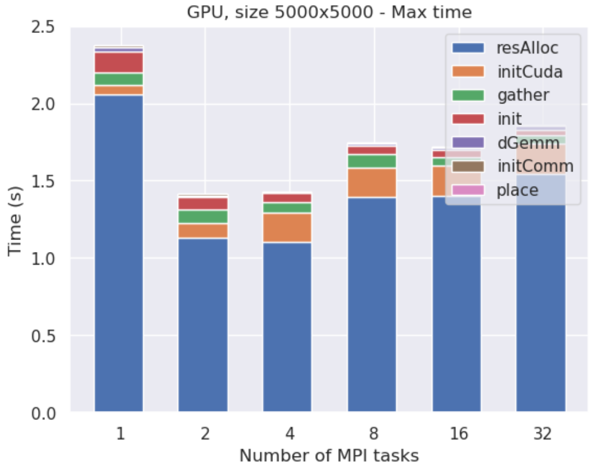


Figure 13: Algorithm with CUBLAS, total execution time

As we can see, the most time consuming part of the code is `resAlloc`, this means we are spending most of the time in moving the matrices from CPU to GPU and back: the matrices are far too small to expect a good speedup from the GPU.

Let's increase the matrix size:

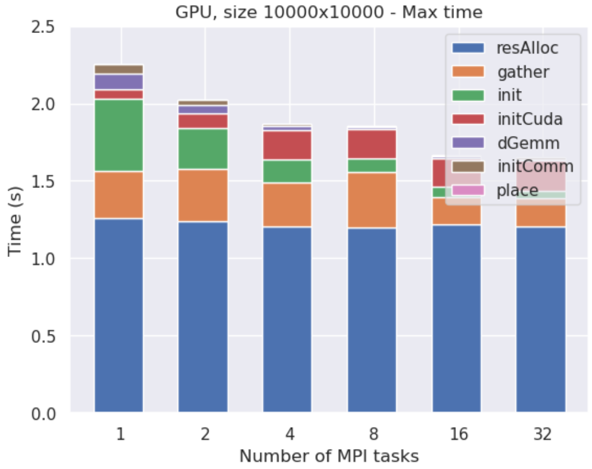


Figure 14: Algorithm with CUBLAS, total execution time

More or less the same results as before, but we can start to appreciate a bit of speedup.

Let's now analyze the results for the 45000x45000 matrices:

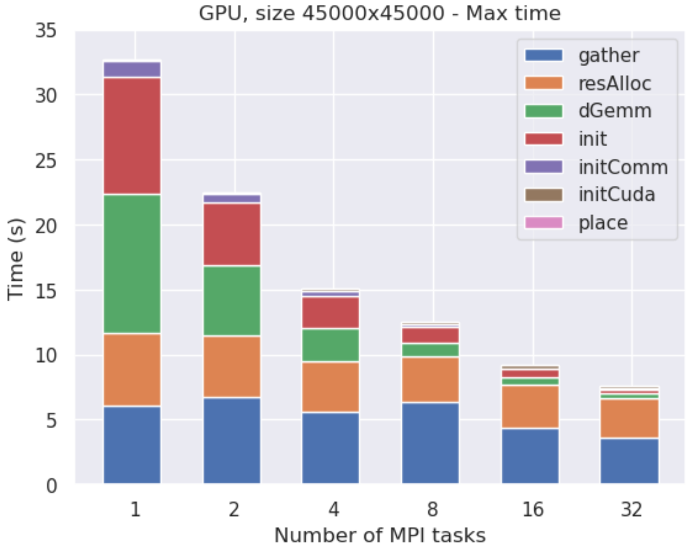


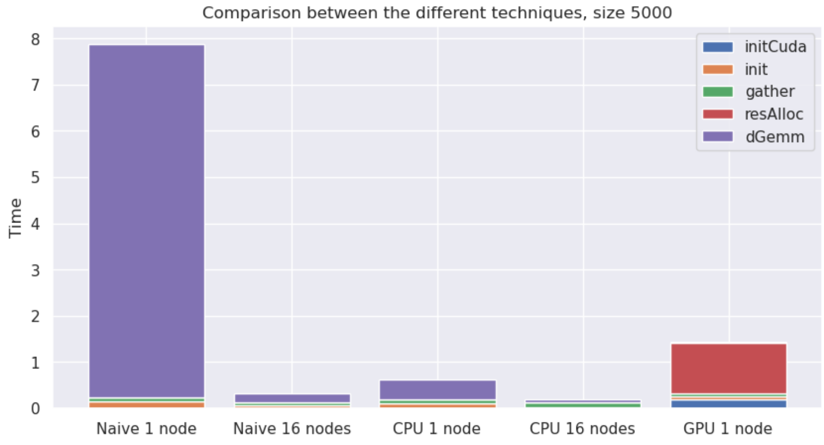
Figure 15: Algorithm with CUBLAS, total execution time

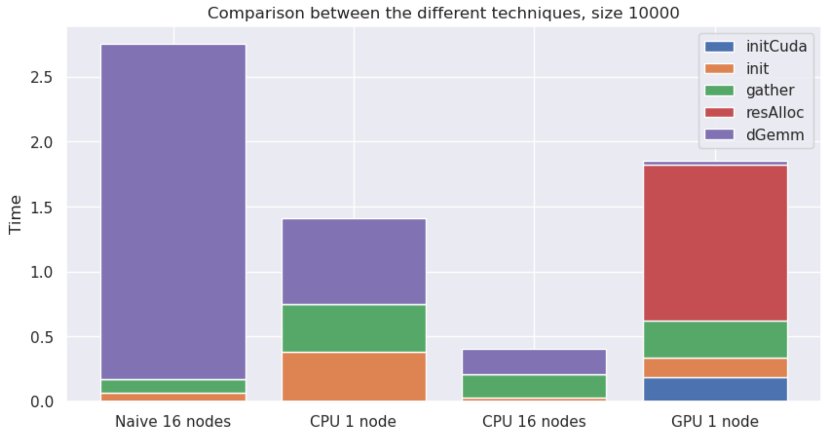
By looking at the results of the measurements, we immediately spot two things:

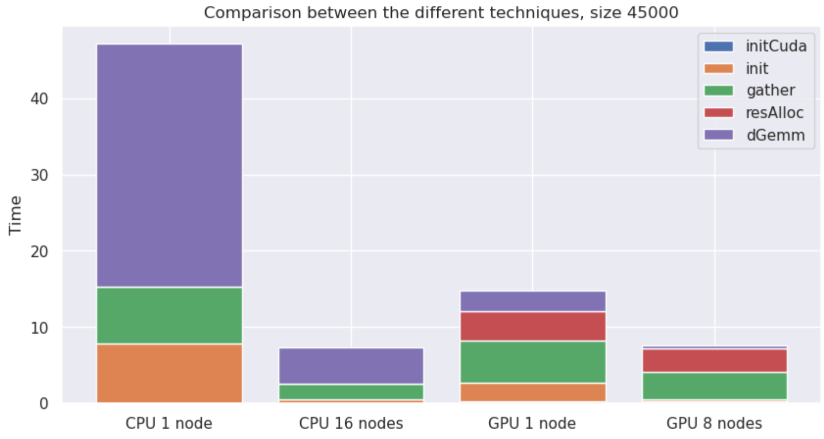
- `init` and `dGemm` take more or less the same time, although the latter is much more computationally intensive than the former, since `init` is performed on the CPU, while `dGemm` is performed on the GPU;
- `gather` and `resAlloc` are still quite relevant, especially with 16 and 32 MPI tasks (corresponding to 4 and 8 nodes).

1.6.4. Comparison

Let’s compare the results obtained by the three algorithms:







1.7. How to run

A Makefile is provided to easily compile and run the code. The available targets are:

- `make naive`: produce an executable running with the naive algorithm (triple loop);
- `make cpu`: produce an executable running with the BLAS library (requires BLAS, on Leonardo you can load it with `module load openblas/0.3.24--nvhpc--23.11`);

> ⚠ Leonardo also provides the `openblas/0.3.24--gcc--12.2.0` module, but this version is not able to execute `cblas_dgemm` routine in parallel, hence we wouldn't be able to exploit the full potential of the CPU.

- `make gpu`: produce an executable running with CUDA and CUBLAS library (requires CUDA and CUBLAS, in Leonardo they are included in the `nvhpc` module which also provides a CUDA-Aware MPI library);
- `make clean`: remove all the executables and the object files.

After compilation, the executables can be run with `mpirun -np <np> ./main <size>`.

The Makefile also provides some shortcuts to directly compile and run the code:

- `make naiverun NP=<np> SZ=<size>`: equivalent to
  `make clean && make naive && mpirun -np <np> ./main <size>`;
- `make cpurun NP=<np> SZ=<size>`: equivalent to
  `make clean && make cpu && mpirun -np <np> ./main <size>`;
- `make gpurun NP=<np> SZ=<size>`: equivalent to
  `make clean && make gpu && mpirun -np <np> ./main <size>`.

2. Jacobi's Algorithm with OpenACC

2.1. Introduction

The second assignment consists of implementing the Jacobi's method to solve Laplace equation in a distributed memory environment, using the MPI library to communicate between processes and OpenACC to parallelize the computation on GPU. The program is expected to run entirely on GPU, without any data transfer between CPU and GPU in the middle of the computation.

Before digging into the implementation of the algorithm, let's first describe the problem and how to solve it.

2.2. Jacobi's algorithm

Laplace's equation is a second-order partial differential equation, often written in the form

$$\nabla^2 V = 0$$

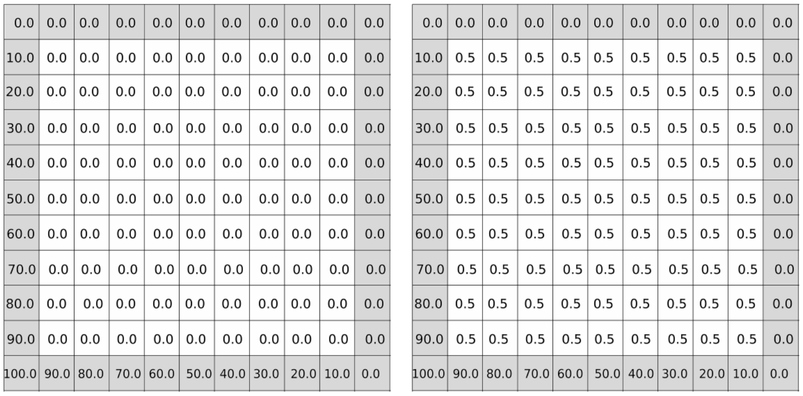
where V is the unknown function of the spatial coordinates x, y, and z, and ∇^2 is the Laplace operator. The Laplace equation is named after Pierre-Simon Laplace, who first studied its properties. Solutions of Laplace's equation are called harmonic functions and are important in many areas of physics, including the study of electromagnetic fields, heat conduction and fluid dynamics. In two dimensions, Laplace's equation is given by

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

whose solution can be iteratively found through Jacobi's method: if we discretize the domain in a grid of points, the value of each point can be updated as the average of its neighbors.

The algorithm works as follows:

- initialize two matrices as in the following picture: the first matrix is filled with zeros, the second one with 0.5, both with the same boundary conditions: 0 in the upper and right boundaries, 100 in the lower left corner, with increasing values starting from that corner and getting farther from it along the left and lower boundaries:



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
20.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
30.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
40.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
50.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
60.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
70.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
80.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
90.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

Figure 16: Matrices initialization

- Iterate over the grid points, updating each internal point of the first matrix as the average of its neighbors in the second matrix:

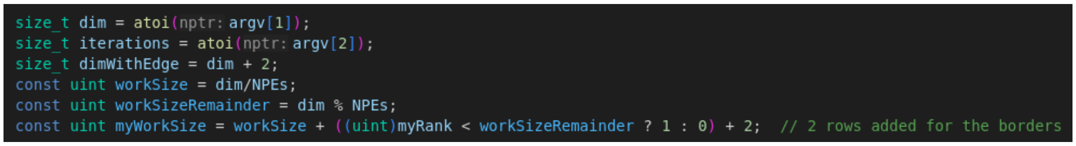
$$V_{i,j}^{k+1} = \frac{1}{4}\left(V_{i-1,j}^{k} + V_{i+1,j}^{k} + V_{i,j-1}^{k} + V_{i,j+1}^{k}\right)$$

- Swap the pointers of the two matrices and repeat points 2 and 3 until a desired convergence criterion is met.

2.3. Distribute the domain: MPI

Since at each iteration each point is updated independently on the others (we only need their old value, which is constant during the update), this algorithm clearly opens the door to parallelization: each process can be assigned a subgrid of the domain, and the communication between processes is only needed at the boundaries of the subgrids.

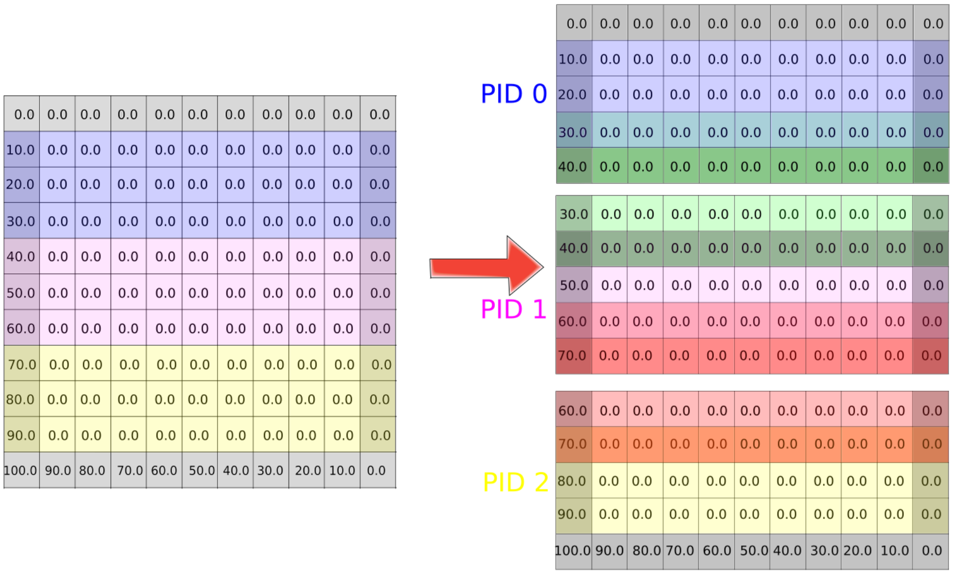
In this assignment, we will consider the domain to be distributed by rows among multiple MPI processes, hence each process will have a subgrid with a fixed number of rows of the entire grid (equal to the total number of rows divided by the number of processes, plus two more rows, one above and one below, that will be needed to perform the update). Since in general the number of rows of the grid is not divisible by the number of processes, some processes will actually have one more row than the others:



```
size_t dim = atoi(nptr: argv[1]);
size_t iterations = atoi(nptr: argv[2]);
size_t dimWithEdge = dim + 2;
const uint workSize = dim/NPEs;
const uint workSizeRemainder = dim % NPEs;
const uint myWorkSize = workSize + ((uint)myRank < workSizeRemainder ? 1 : 0) + 2;  // 2 rows added for the borders
```

Figure 17: How different processes share the work

For example, if `dim`= 9 and `NPEs`= 3, we have the situation showed in the following picture:



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

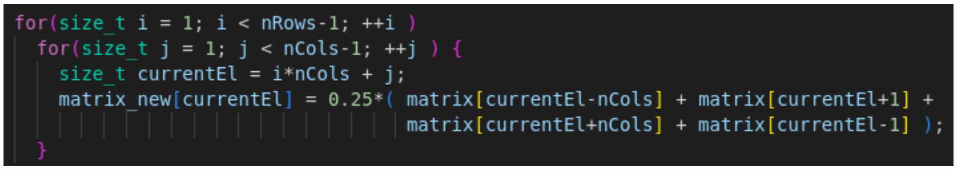
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

Figure 18: Rows exchange between processes

The idea to compute the solution is the following: each process will have two submatrices with `myWorkSize = 9/3 + 0 + 2 = 5` rows, also considering the 2 ghost rows, one above and one below, to perform the update. Each process only initializes and updates the internal rows of one submatrix, and then exchanges the boundary rows with the neighbor processes.

More precisely, each process first initializes its own submatrices and then continuously:

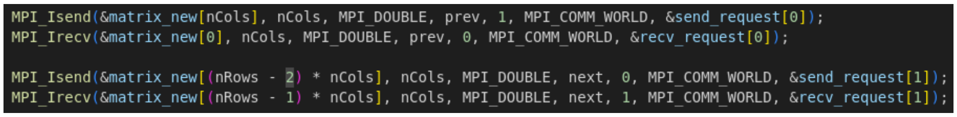
- updates the values of the internal points (hence excluding its first and last row and first and last column) of one submatrix using the values from the other one:



```
for(size_t i = 1; i < nRows-1; ++i )
  for(size_t j = 1; j < nCols-1; ++j ) {
    size_t currentEl = i*nCols + j;
    matrix_new[currentEl] = 0.25*( matrix[currentEl-nCols] + matrix[currentEl+1] +
                                   matrix[currentEl+nCols] + matrix[currentEl-1] );
  }
```

Figure 19: Matrix update

- updates the boundaries: it sends its second row to the upper process and its semilast row to the lower one, and receives its first row from the upper process and its last row from the lower one (first and last process only send and receive a single row, since the other one is a fixed boundary condition):



```
MPI_Isend(&matrix_new[nCols], nCols, MPI_DOUBLE, prev, 1, MPI_COMM_WORLD, &send_request[0]);
MPI_Irecv(&matrix_new[0], nCols, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD, &recv_request[0]);

MPI_Isend(&matrix_new[(nRows - 2) * nCols], nCols, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &send_request[1]);
MPI_Irecv(&matrix_new[(nRows - 1) * nCols], nCols, MPI_DOUBLE, next, 1, MPI_COMM_WORLD, &recv_request[1]);
```

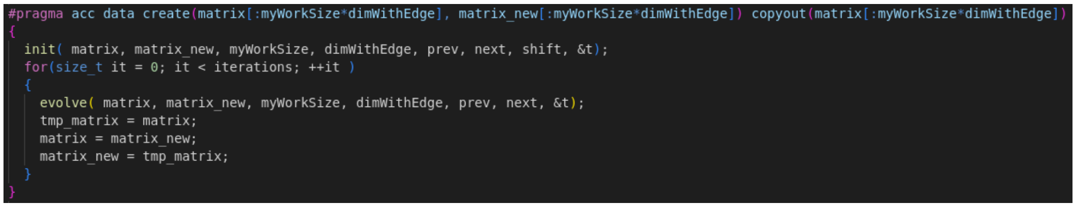
Figure 20: Rows exchange between processes

- swaps the pointers to the matrices, so that the new matrix becomes the old one and vice versa;

until a desired convergence criterion is met.

2.4. Move to GPU: OpenACC

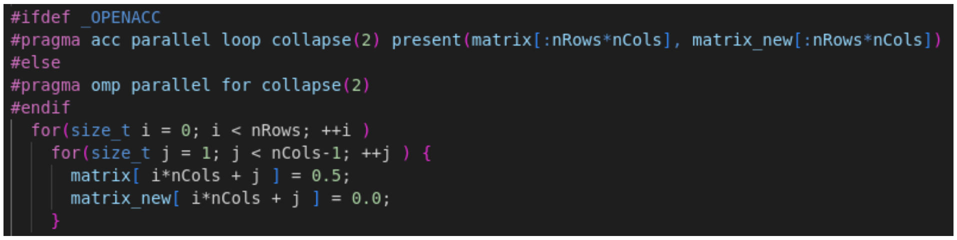
The Jacobi's method is a perfect candidate for GPU acceleration, and OpenACC offers simple and powerful instruments to do so. The idea is to generate a `data` region to allocate the matrix on the GPU and perform both initialization and updates there, and then copy it back to the CPU:



```
#pragma acc data create(matrix[:myWorkSize*dimWithEdge], matrix_new[:myWorkSize*dimWithEdge]) copyout(matrix[:myWorkSize*dimWithEdge])
{
  init( matrix, matrix_new, myWorkSize, dimWithEdge, prev, next, shift, &t);
  for(size_t it = 0; it < iterations; ++it )
  {
    evolve( matrix, matrix_new, myWorkSize, dimWithEdge, prev, next, &t);
    tmp_matrix = matrix;
    matrix = matrix_new;
    matrix_new = tmp_matrix;
  }
}
```

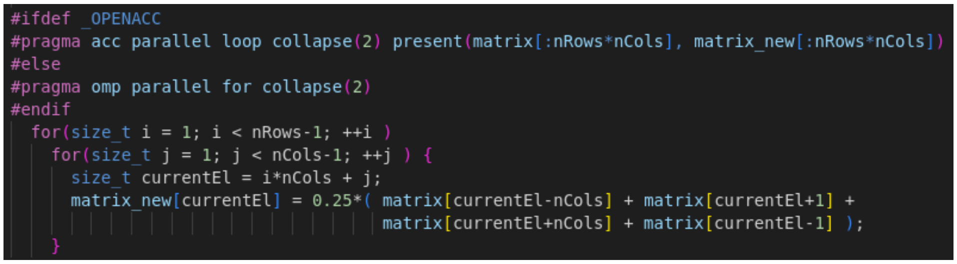
Figure 21: Open a data region

Both initialization and update can then be parallelized using the `parallel loop` directive:



```
#ifdef _OPENACC
#pragma acc parallel loop collapse(2) present(matrix[:nRows*nCols], matrix_new[:nRows*nCols])
#else
#pragma omp parallel for collapse(2)
#endif
  for(size_t i = 0; i < nRows; ++i )
    for(size_t j = 1; j < nCols-1; ++j ) {
      matrix[ i*nCols + j ] = 0.5;
      matrix_new[ i*nCols + j ] = 0.0;
    }
```

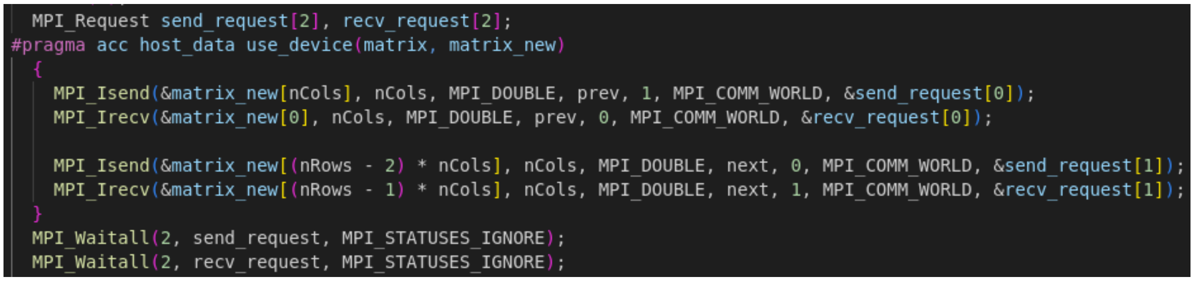
Figure 22: A part of matrices initialization, done with OpenACC



```
#ifdef _OPENACC
#pragma acc parallel loop collapse(2) present(matrix[:nRows*nCols], matrix_new[:nRows*nCols])
#else
#pragma omp parallel for collapse(2)
#endif
  for(size_t i = 1; i < nRows-1; ++i )
    for(size_t j = 1; j < nCols-1; ++j ) {
      size_t currentEl = i*nCols + j;
      matrix_new[currentEl] = 0.25*( matrix[currentEl-nCols] + matrix[currentEl+1] +
                                     matrix[currentEl+nCols] + matrix[currentEl-1] );
    }
```

Figure 23: Matrix update with OpenACC

Also, in order to execute the the rows exchange directly between GPUs, `acc host_data use_device` directive has been used:



```
  MPI_Request send_request[2], recv_request[2];
#pragma acc host_data use_device(matrix, matrix_new)
  {
    MPI_Isend(&matrix_new[nCols], nCols, MPI_DOUBLE, prev, 1, MPI_COMM_WORLD, &send_request[0]);
    MPI_Irecv(&matrix_new[0], nCols, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD, &recv_request[0]);

    MPI_Isend(&matrix_new[(nRows - 2) * nCols], nCols, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &send_request[1]);
    MPI_Irecv(&matrix_new[(nRows - 1) * nCols], nCols, MPI_DOUBLE, next, 1, MPI_COMM_WORLD, &recv_request[1]);
  }
  MPI_Waitall(2, send_request, MPI_STATUSES_IGNORE);
  MPI_Waitall(2, recv_request, MPI_STATUSES_IGNORE);
```

Figure 24: Rows exchange between processes, performed on GPUs

2.5. Results

In this section we will analyze the performances obtained by the algorithm, both on CPU and on GPU. The code has been run on the Leonardo cluster, with up to 16 MPI tasks allocated one per node, for CPU versions, and up to 32 MPI tasks allocated four per node, one per GPU card, for the GPU version. The execution time has been measured with the `MPI_Wtime` function. The tests have been done with a matrix of size 1200x1200 and 12000x12000, with 10 evolution iterations, and 40000x40000, with 1000 iterations, to better study the scalability. The maximum time among all the MPI processes has been plotted. However, I have also collected data regarding the average time and they have showed the same behavior, meaning the workload is correctly distributed among the processes, for this reason they have not been plotted.

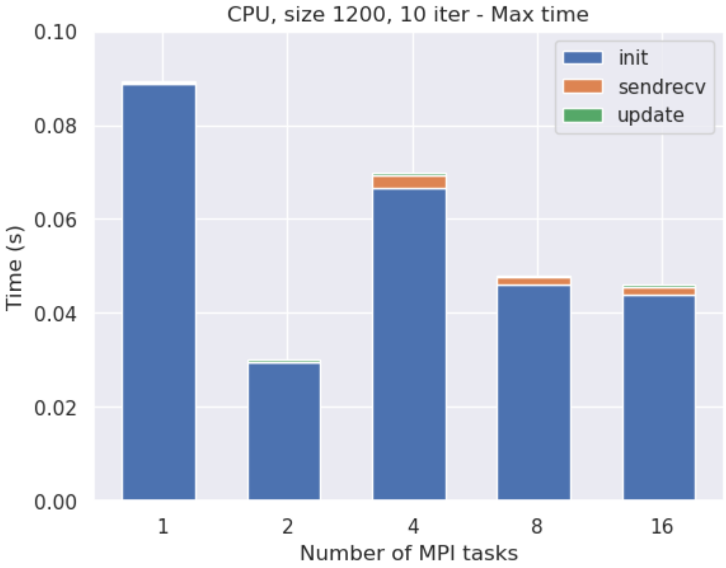
To easily identify the different parts of the code and plot them I have used some terms, here a brief explanation of them is given, in order of appearance in the code:

- `initacc`: initialization of OpenACC, with `acc_get_num_devices`, `acc_set_device_num` and `acc_init`;
- `copyin`: enter the data region, allocate the matrices on GPU;
- `init`: initialization of the matrices;
- `update`: total time spent on updating the matrix;
- `sendrecv` total time spent on exchanging the ghost rows;
- `save`: save the matrix on file using MPI-IO;
- `copyout`: copying the output matrix from GPU to CPU.

> **ℹ Note**
>
> To further improve performances on CPU, OpenMP is used to parallelize both the initialization and the update of the matrices when GPUs are not available.

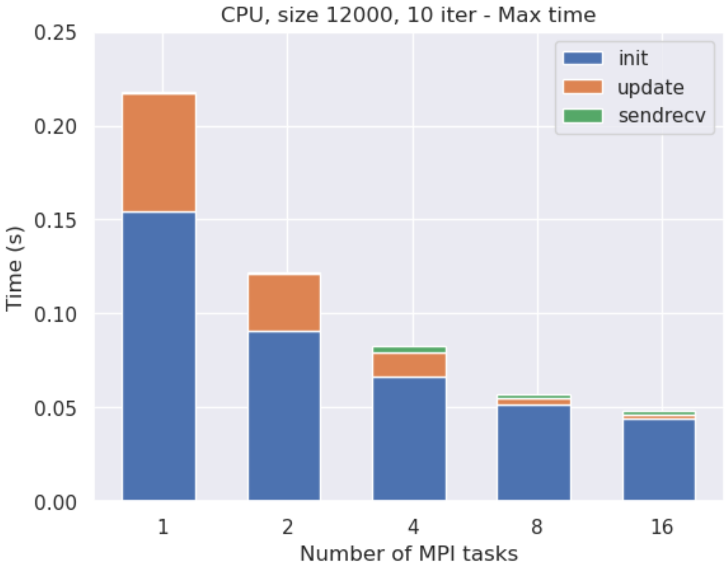
2.5.1. CPU

Let's start with the CPU version:



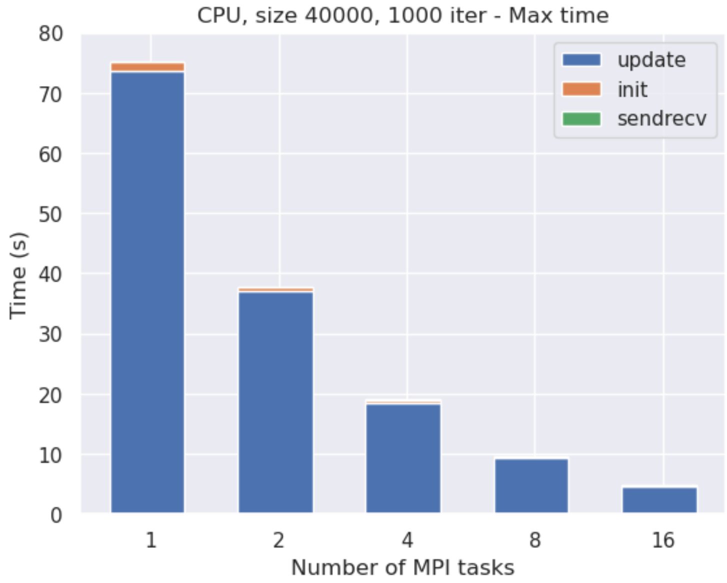
As we can see, there is basically no scalability due to the very low time spent. `init` takes almost all the time, with `update` being practically irrelevant due to the very low number of updates done.

Let's see how things change with a larger matrix:



We can start to appreciate some speedup, and the time spent on `update` is now relevant, although `init` is still the most time-consuming part of the code, but scalability still almost interrupts after 4 tasks.

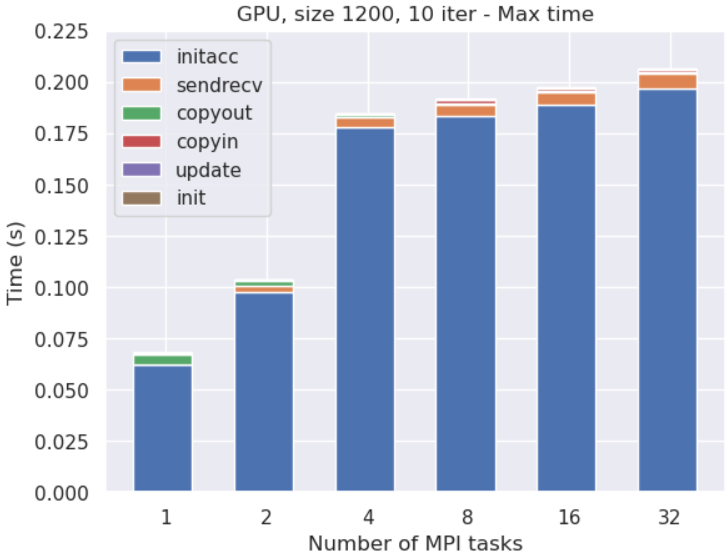
Let's see what happens with a much larger matrix and more iterations:



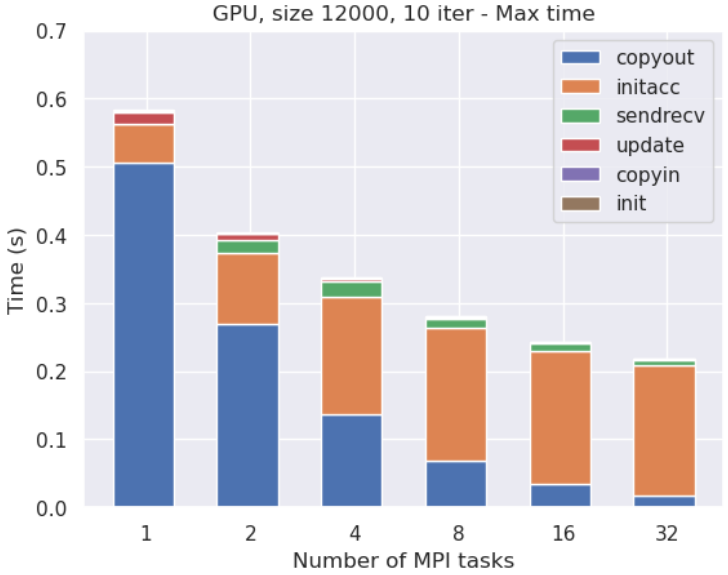
We can finally appreciate a great scalability, with the time spent on `update` being the most relevant part of the code, as we would expect.

2.5.2. GPU

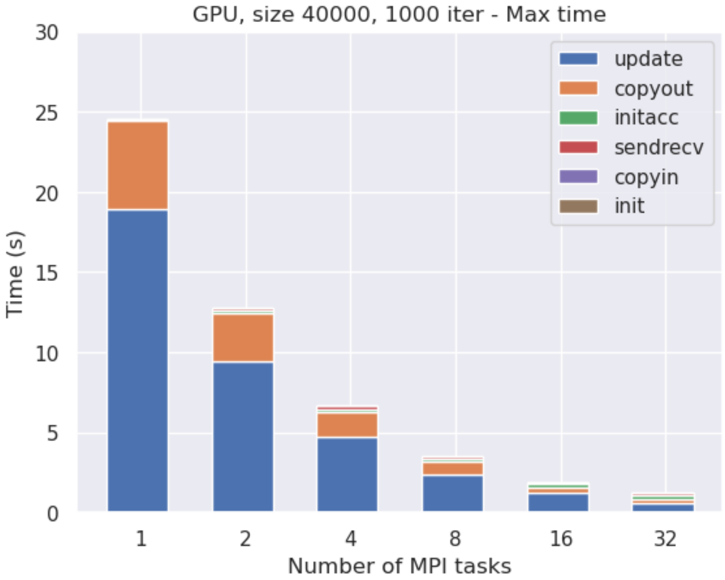
Let's now move to the GPU version:



As we can see, it is totally pointless to run the code on GPU with such a small matrix, since most of the time is spent on `initacc`, hence we would get no speedup at all. Let's see what happens with a larger matrix:



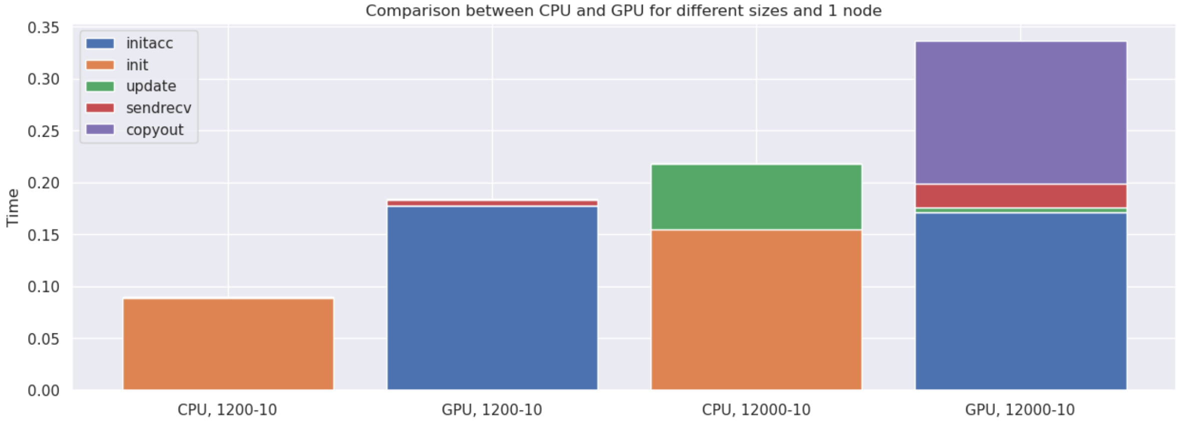
We can start to appreciate some speedup, but the time spent on `initacc` is still relevant and most of the time is spent in `copyout` (with less tasks) or still in `initacc` (with more tasks), with `init` and `update` being basically negligible. This is due to the fact that the workload is too small to fully exploit the power of the GPU. Let's then try to run the code with a much larger matrix and many more iterations, in order to increase the workload:

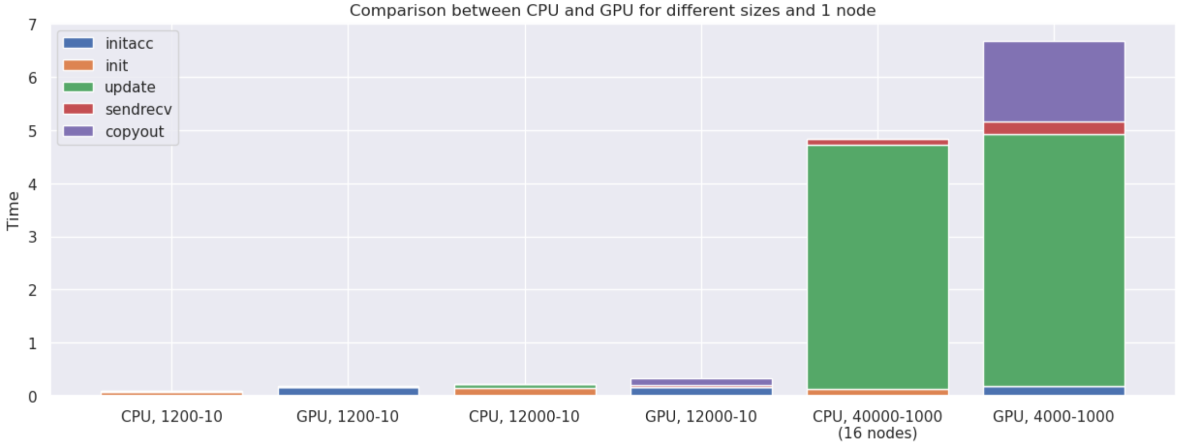


We can finally appreciate a significant speedup even with a large number of MPI tasks: the time spent on `initacc` is now negligible with respect to the other parts of the code and most of the time is now spent on the update, as we would expect.

2.5.3. Comparison

Let's now compare the CPU and GPU versions with the same matrix size and number of iterations:

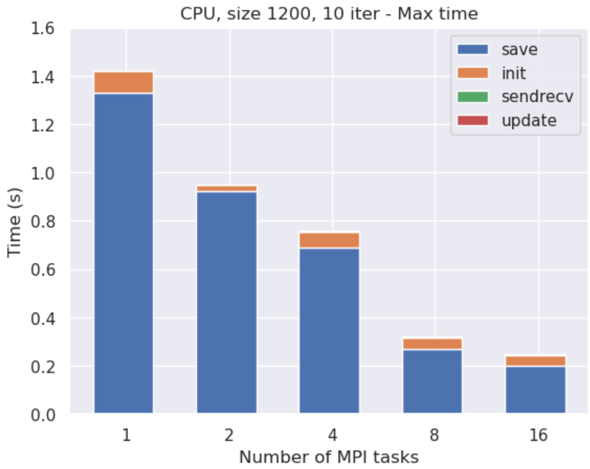


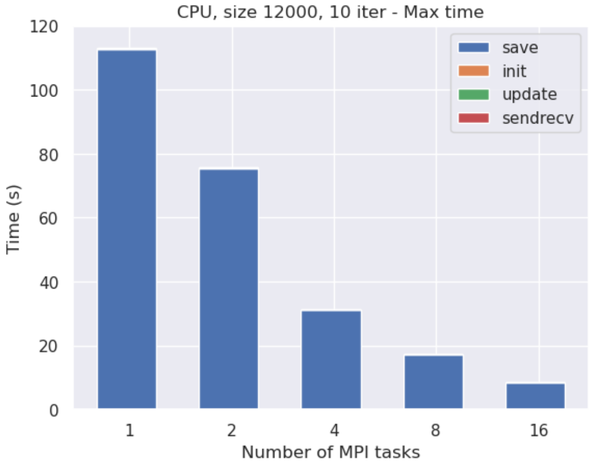


As we can see, if with a small matrix the GPU version is not convenient at all, with a larger matrix we can appreciate a significant boost in performances, with the execution time for a single node being comparable with the one with 16 CPU nodes.

2.5.4. Save time

Up to now we have ignored the `save` time, let's now see how it affects the performances: since it is not influenced by GPU acceleration, we'll just compare it with other parts of the code in order to understand its magnitude:





As we can see, using MPI-IO we are able to save some time writing on file in parallel, but this is still by far the most time-consuming part of the code.

2.6. How to run

> Tip
>
> `%pu...` means that the target exists both as `cpu...` and `gpu...`

A Makefile is provided to easily compile and run the code. The available targets are:

- `make %pu`, `make %pusave` and `make %pugif`: produce an executable running on CPU with OpenMP or on GPU with OpenACC, the second one also saves the final matrix in a file `solution.dat`, while the third one saves the entire evolution of the matrix in multiple `.dat` files;
- `make plot`: produce a plot using Gnuplot: if the code has been compiled with the `save` option, it will plot the final matrix in a file `solution.png`, while with the `gif` option it will plot a gif with the evolution of the matrix in a file `solution.gif`;
- `make clean`: remove all the executables and the object files.

After compilation, the executables can be run with `mpirun -np <np> ./jacobi.x <size> <nIter>`.

The Makefile also provides a shortcut to directly compile and run the code and save the output:

```
make %purun NP=<np> SZ=<size> IT=<nIter>
```

equivalent to

```
make clean && make %pusave && mpirun -np NP ./jacobi.x SZ IT && make plot
```

2.6.1. Check correctness

In order to check correctness of the obtained output, the original serial code is provided in original_code folder, and a special target can be used to directly compare the output of the original code with the one of the optimized code:

```
make compare%pu NP=<nProc> SZ=<size> IT=<nIter>
```

This target will compile and run both the original and the optimized code (with the given number of processes, size and number of iterations, on CPU or GPU), save the outputs in binary format, and compare them using Unix command `diff`: if the outputs are identical, as expected, no output will be produced, otherwise the output will be

```
Binary files output/solution0.dat and original_code/solution.dat differ
```

> ℹ Note
>
> To directly compare the two files without having to worry about precision issues, the original code `save_gnuplot` function has been modified to save binary files; this is the only change that has been performed on it.

> ⚠ Warning
>
> MPI-IO writes binary files and does not truncate the file on which it'll write if it already exists: if you want to run the program with a size which is smaller than the previous one, `make clean` or empty the `output` folder before running, in order to generate it from scratch instead of overwriting it. `compare%pu` targets are already provided with an internal `clean`, in order to repeatedly compare results without having to worry about non-truncated files.

3. Jacobi's Algorithm with One-Sided MPI

3.1. Introduction

The third assignment consists of implementing the Jacobi's method to solve Laplace equation in a distributed memory environment, using the MPI library to communicate between processes in a one-sided fashion.

Before digging into the implementation of the algorithm, let's first describe the problem and how to solve it.

3.2. Jacobi's algorithm

Laplace's equation is a second-order partial differential equation, often written in the form

$$\nabla^2 V = 0$$

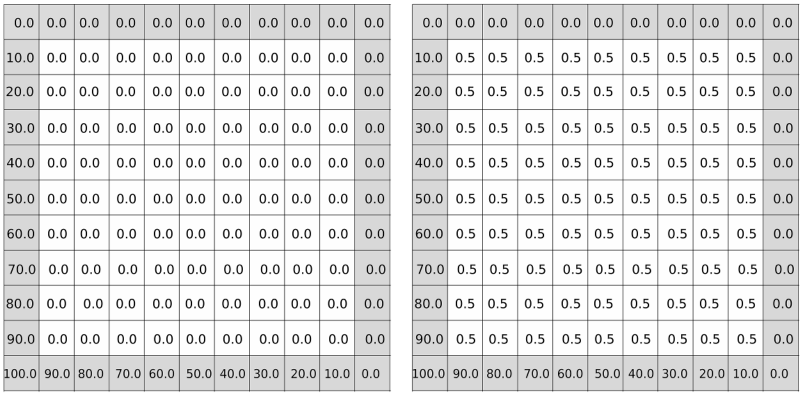
where V is the unknown function of the spatial coordinates x, y, and z, and ∇^2 is the Laplace operator. The Laplace equation is named after Pierre-Simon Laplace, who first studied its properties. Solutions of Laplace's equation are called harmonic functions and are important in many areas of physics, including the study of electromagnetic fields, heat conduction and fluid dynamics. In two dimensions, Laplace's equation is given by

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

whose solution can be iteratively found through Jacobi's method: if we discretize the domain in a grid of points, the value of each point can be updated as the average of its neighbors.

The algorithm is as follows:

- initialize two matrices as in the following picture: the first matrix is filled with zeros, the second one with 0.5, both with the same boundary conditions: 0 in the upper and right boundaries, 100 in the lower left corner, with increasing values starting from that corner and getting farther from it along the left and lower boundaries:



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
20.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
30.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
40.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
50.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
60.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
70.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
80.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
90.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

Figure 25: Matrices initialization

- Iterate over the grid points, updating each internal point of the first matrix as the average of its neighbors in the second matrix:

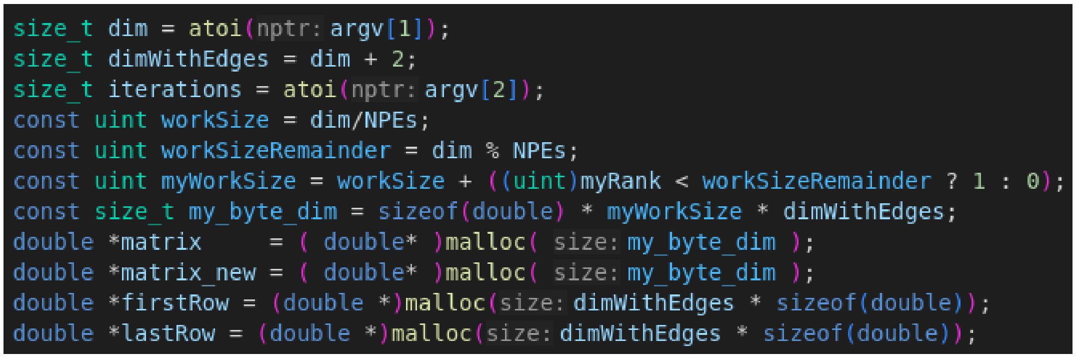
$$V_{i,j}^{k+1} = \frac{1}{4}\left(V_{i-1,j}^{k} + V_{i+1,j}^{k} + V_{i,j-1}^{k} + V_{i,j+1}^{k}\right)$$

- Swap the pointers of the two matrices and repeat points 2 and 3 until a desired convergence criterion is met.

3.3. Distribute the domain: MPI

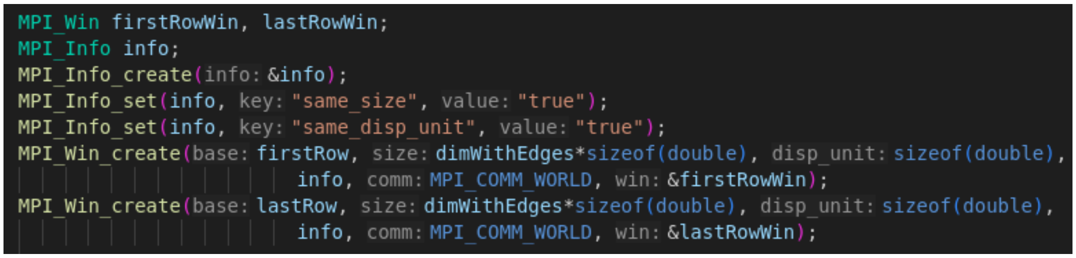
Since at each iteration each point is updated independently on the others (we only need their old value, which is constant during the update), this algorithm clearly opens the door to parallelization: each process can be assigned a subgrid of the domain, and the communication between processes is only needed at the boundaries of the subgrids.

In this assignment, we will consider the domain to be distributed by rows among multiple MPI processes, hence each process will have a subgrid with a fixed number of rows of the entire grid (equal to the total number of rows divided by the number of processes), and **two more rows**, needed to perform the update, open for the other processes to access and update them through the use of two `MPI_Win` objects. Since in general the number of rows of the grid is not divisible by the number of processes, some processes will actually have one more row than the others:



```
size_t dim = atoi(nptr: argv[1]);
size_t dimWithEdges = dim + 2;
size_t iterations = atoi(nptr: argv[2]);
const uint workSize = dim/NPEs;
const uint workSizeRemainder = dim % NPEs;
const uint myWorkSize = workSize + ((uint)myRank < workSizeRemainder ? 1 : 0);
const size_t my_byte_dim = sizeof(double) * myWorkSize * dimWithEdges;
double *matrix     = ( double* )malloc( size: my_byte_dim );
double *matrix_new = ( double* )malloc( size: my_byte_dim );
double *firstRow = (double *)malloc(size: dimWithEdges * sizeof(double));
double *lastRow = (double *)malloc(size: dimWithEdges * sizeof(double));
```

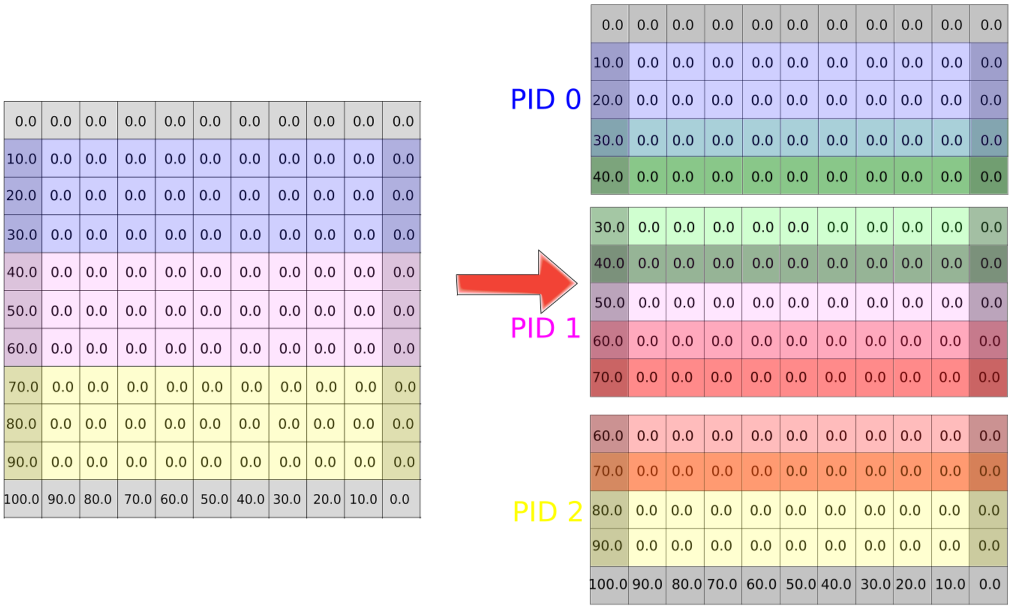
Figure 26: How different processes share the work



```
MPI_Win firstRowWin, lastRowWin;
MPI_Info info;
MPI_Info_create(info: &info);
MPI_Info_set(info, key: "same_size", value: "true");
MPI_Info_set(info, key: "same_disp_unit", value: "true");
MPI_Win_create(base: firstRow, size: dimWithEdges*sizeof(double), disp_unit: sizeof(double),
                        info, comm: MPI_COMM_WORLD, win: &firstRowWin);
MPI_Win_create(base: lastRow, size: dimWithEdges*sizeof(double), disp_unit: sizeof(double),
                        info, comm: MPI_COMM_WORLD, win: &lastRowWin);
```

Figure 27: Windows generation

For example, if `dim`= 9 and `NPEs`= 3, we have the situation showed in the following picture:



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

PID 0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

PID 1

30.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

PID 2

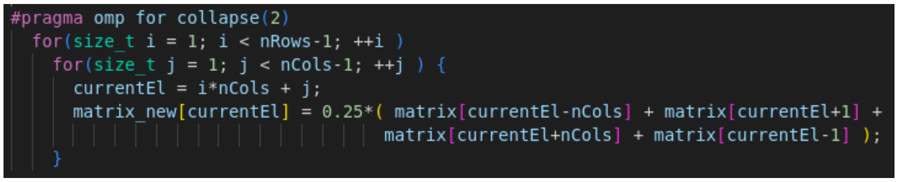
60.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100.0	90.0	80.0	70.0	60.0	50.0	40.0	30.0	20.0	10.0	0.0

Figure 28: Rows exchange between processes

The idea to compute the solution is the following: each process has two submatrices with `myWorkSize = 9/3 + 0 = 3` rows, and 2 more rows to perform the update. Each process only initializes and updates one submatrix and then puts its first and last row inside the neighbor processes' windows.

More precisely, each process first initializes its own submatrices and its extra rows, and then continuously:

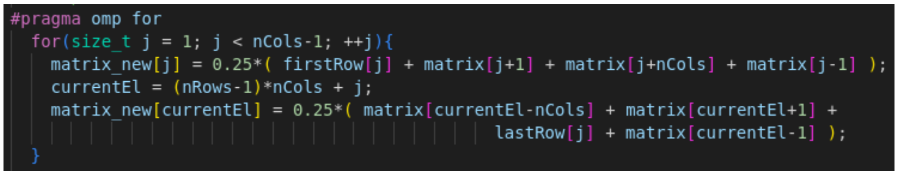
- updates the values of the internal points of one submatrix (hence excluding its first and last row and the first and last column) using the values from the other one:



```
#pragma omp for collapse(2)
  for(size_t i = 1; i < nRows-1; ++i )
    for(size_t j = 1; j < nCols-1; ++j ) {
      currentEl = i*nCols + j;
      matrix_new[currentEl] = 0.25*( matrix[currentEl-nCols] + matrix[currentEl+1] +
                                     matrix[currentEl+nCols] + matrix[currentEl-1] );
    }
```

Figure 29: Matrix update

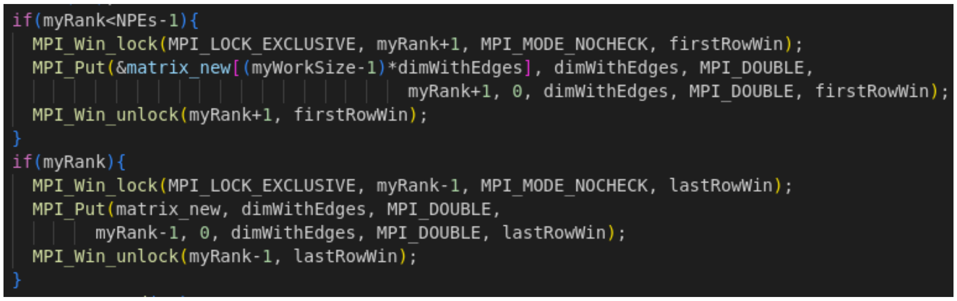
- updates the first and last row of the submatrix, using the other one and the extra rows:



```
#pragma omp for
  for(size_t j = 1; j < nCols-1; ++j){
    matrix_new[j] = 0.25*( firstRow[j] + matrix[j+1] + matrix[j+nCols] + matrix[j-1] );
    currentEl = (nRows-1)*nCols + j;
    matrix_new[currentEl] = 0.25*( matrix[currentEl-nCols] + matrix[currentEl+1] +
                                                lastRow[j] + matrix[currentEl-1] );
  }
```

Figure 30: Extra rows update

- puts the first row of the submatrix inside the upper process' second window and the last row of the submatrix inside the lower process' first one (first and last process only put a single row, since the other one is a fixed boundary condition):



```
if(myRank<NPEs-1){
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, myRank+1, MPI_MODE_NOCHECK, firstRowWin);
  MPI_Put(&matrix_new[(myWorkSize-1)*dimWithEdges], dimWithEdges, MPI_DOUBLE,
                                      myRank+1, 0, dimWithEdges, MPI_DOUBLE, firstRowWin);
  MPI_Win_unlock(myRank+1, firstRowWin);
}
if(myRank){
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, myRank-1, MPI_MODE_NOCHECK, lastRowWin);
  MPI_Put(matrix_new, dimWithEdges, MPI_DOUBLE,
        myRank-1, 0, dimWithEdges, MPI_DOUBLE, lastRowWin);
  MPI_Win_unlock(myRank-1, lastRowWin);
}
```

Figure 31: Put the updated rows inside the neighbor windows

- swaps the pointers to the matrices, so that the new matrix becomes the old one and vice versa;

until a desired convergence criterion is met.

3.4. Results

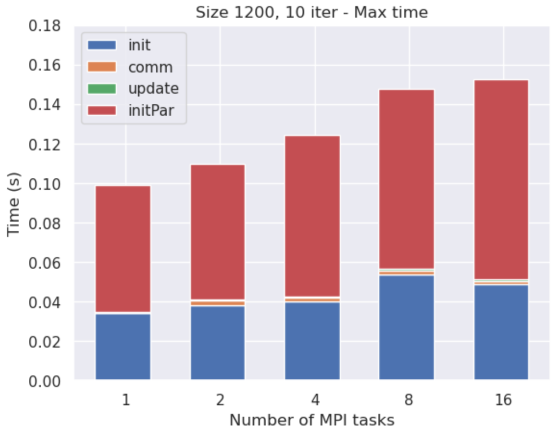
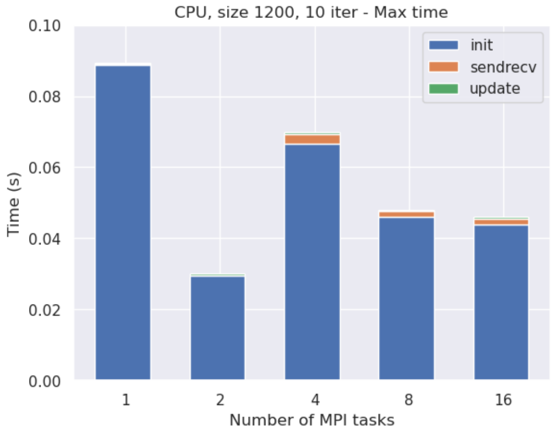
In this section we will analyze the performances obtained by the algorithm. The code has been run on the Leonardo cluster, with up to 16 MPI tasks allocated one per node. The execution time has been measured with the `MPI_Wtime` function. The tests have been done with a matrix of size 1200x1200 and 12000x12000, with 10 evolution iterations, and 40000x40000, with 1000 iterations, to better study the scalability. The maximum time among all the MPI processes has been plotted. However, I have also collected data regarding the average time and they have showed the same behavior, meaning the workload is correctly distributed among the processes, for this reason they have not been plotted.

To easily identify the different parts of the code and plot them I have used some terms, here a brief explanation of them is given, in order of appearance in the code:

- `initPar`: parameters and windows initialization;
- `init`: initialization of the matrices;
- `update`: total time spent on updating the matrix;
- `comm`: total time spent on updating the extra rows;
- `save`: save the matrix on file using MPI-IO.

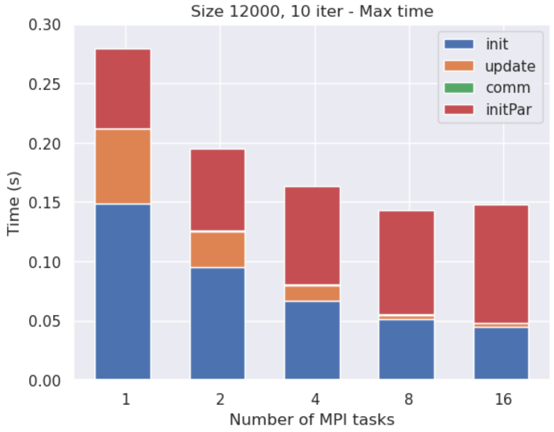
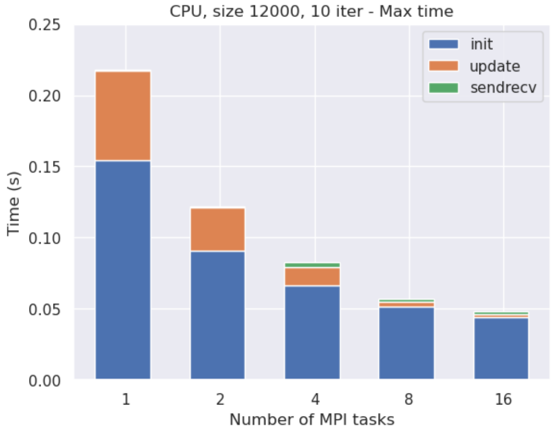
We'll also plot the results obtained with the standard Send/Recv communication, in order to compare the performances of the two methods.

Let's start with the results obtained with the 1200x1200 matrix:

One-sided communication	Send-Recv communication
	

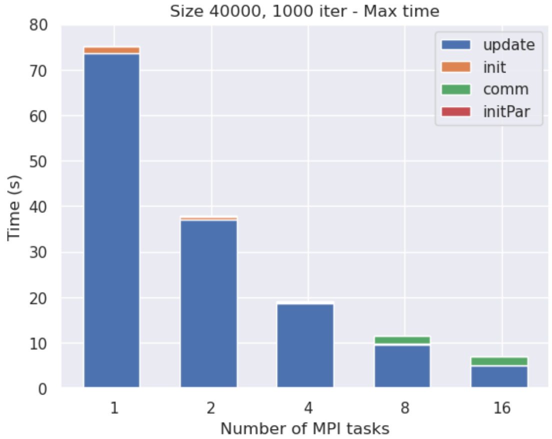
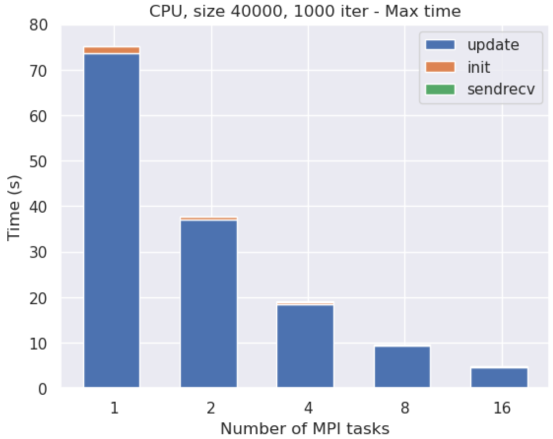
As we can see, there is no scalability due to the very low time spent: `initPar` takes more than half of the total time, and the time spent on `update` is negligible. Similar results were obtained with the standard Send/Recv communication, but in that case `init` was the only relevant part of the code.

Let's see how things change with a larger matrix:

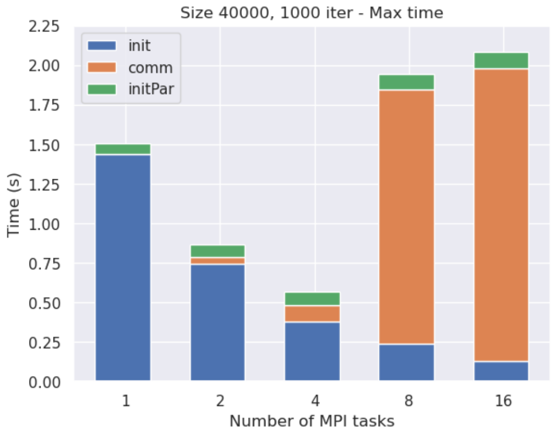
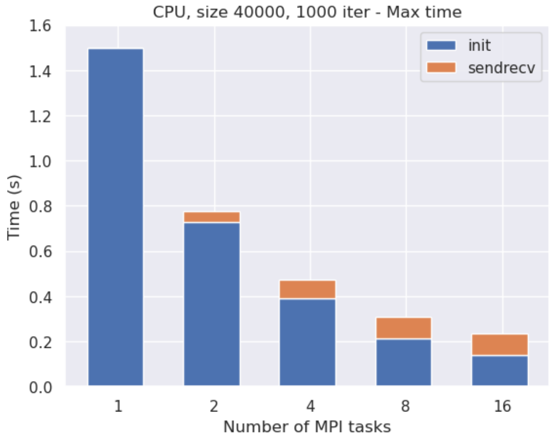
One-sided communication	Send-Recv communication
	

With a larger matrix we can start to appreciate some speedup, and the time spent on `update` is now significant, although `initPar` is still very relevant. We can observe as both the `init` and `update` parts of the code behave very similarly to the standard Send/Recv communication, but in that case the scalability is much better since there is no windows initialization.

Let's see what happens with a much larger matrix and more iterations:

One-sided communication	Send-Recv communication
	

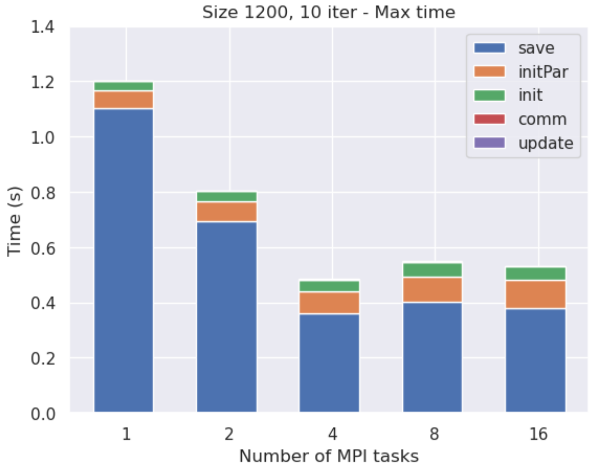
We can finally appreciate a great scalability, with the time spent on `update` being the most relevant part of the code, as we would expect. `update` time is basically the same for both the one-sided and the standard Send/Recv communication, let's see how the other parts behave:

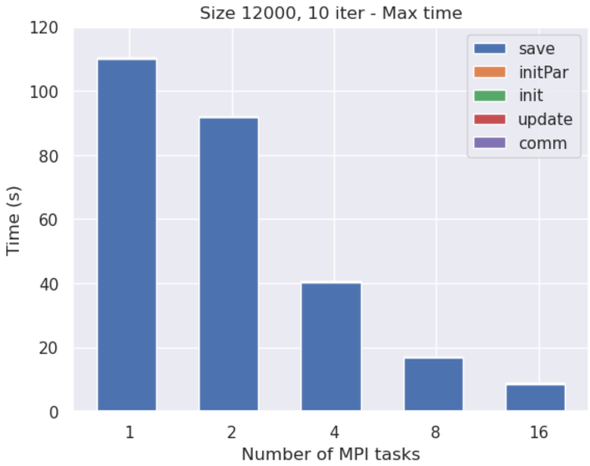
One-sided communication	Send-Recv communication
	

`init` still shows the same behavior in the two cases, while the communication time is far worse with the one-sided communication, especially with higher number of tasks.

3.4.1. Save time

Up to now we have ignored the `save` time, let's now see how it behaves compared to the other parts of the code:





As we can see, using MPI-IO we are able to save some time writing on file in parallel, but the time spent on this part is still by far the most time-consuming part of the code.

3.5. How to run

A Makefile is provided to easily compile and run the code. The available targets are:

- `make`: produce an executable that prints the elapsed times;
- `make save`: produce an executable that also saves the final matrix in a file `solution.dat`;
- `make gif`: produce an executable that also saves the evolution of the matrix in multiple `.dat` files;
- `make plot`: produce a plot using Gnuplot: if the code has been compiled with the `save` target, it will plot the final matrix in a file `solution.png`, while with the `gif` option it will plot a gif with the evolution of the matrix in a file `solution.gif`, both in the `output` folder;
- `make clean`: remove all the executables and the object files.

After compilation, the executables can be run with `mpirun -np <np> ./main <size> <nIter>`.

The Makefile also provides a shortcut to directly compile and run the code and save the output: `make run NP=<np> SZ=<size> IT=<nIter>`, equivalent to `make clean && make save && mpirun -np NP ./jacobi.x SZ IT && make plot`.

3.5.1. Check correctness

In order to check correctness of the obtained output, the serial code is provided in original_code folder, and a special target can be used to directly compare the output of the original code with the one of the optimized code: `make compare NP=<nProc> SZ=<size> IT=<nIter>` This target will compile and run both the original and the optimized code (with the given number of processes, size and number of iterations), save the outputs in binary format, and compare them using Unix command `diff`: if the outputs are identical, as expected, no output will be produced, otherwise the output will be

```
Binary files output/solution0.dat and original_code/solution.dat differ
```

> ⚠ Warning
>
> MPI-IO writes binary files and does not truncate the file on which it'll write if it already exists: if you want to run the program with a size which is smaller than the previous one, delete the `solution.dat` file before running, in order to generate it from scratch instead of overwriting it. `compare` target is already provided with an internal `clean`, in order to repeatedly compare results without having to worry about non-truncated files.