FINAL EXAM REPORT

Course in Advanced HPC

Paolo Vizzo
DSSC - University Of Trieste
July 2024

Contents

1 Introduction

In this report are presented some strong scaling results on the Leonardo Cluster involving two type of parallel algorithm: the double precision matrix multiplication and the Jacobi iteration method to solve the 2D heat equation. The scalings were performed both on the CPU and and GPU nodes. The library used are MPI, OpenMP, Intel MKL, OpenBLAS, Cuda and OpenACC.

2 Scaling Setup

2.1 CPU Nodes

- 1, 2, 4, 8, 16 nodes
- 1 MPI Task per node
- 112 OpenMP threads per node (1 per core)

2.2 GPU Nodes

- 1, 2, 4, 8, 16 nodes
- 4 MPI Task per node (1 per GPU)
- 8 OpenMP threads per task (32 per node, 1 per core)

Each run was repeated 5 times in order to calculate the averages (except for Naive MatMul).

Part I

Parallel Matrix Multiplication

3 Algorithm Implementation

3.1 CPU version

To perform the parallel matrix multiplication $C = AB$ on CPU we start by allocating and initializing two matrices of size $N \times N$: the identity matrix A and the matrix B. Each process allocates its local portion for each matrix A_{LOC} and B_{LOC}. Moreover we allocate for each process:

- B_{TEMP} of size $(N/nprocs + 1) \times N_{LOC}$, the block of matrix B that a process send with *AllGather* operation
- B_{TEMP_N} of size $(N/nprocs + 1) \times N$, the array that stores the result of *AllGather* operation for each process.
- C_{TEMP_N} of size $(N/nprocs + 1) \times N$, which stores the results (by row) of the multiplication for each process.

N_{LOC} is the number of rows assigned to each process, *nprocs* the number of processes and $(N/nprocs + 1)$ is the maximum number of column of matrix B that a process can have at each iteration of matrix multiplication.

Then, the iterations from 0 to *nprocs* works for each process as shown in Algorithm 1 and illustrated in Figure 1.

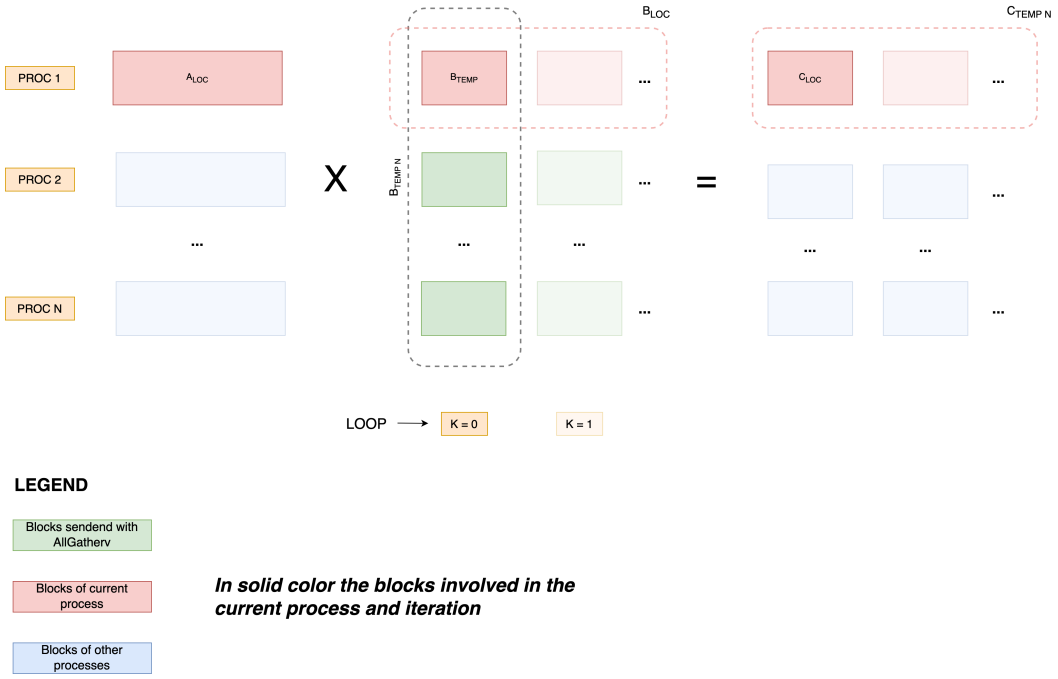


Figure 1: Iteration for one process in Parallel MatMul Algorithm

Algorithm 1 Parallel Matrix Multiplication Iteration - CPU

1: **for** $k \leftarrow 0$ to *nprocs* **do**
2: **if** $k <$ rest **then**
3: $N_{COL} \leftarrow N/nprocs + 1$
4: C_{LOC} points to $C_{TEMP\_N} + (N_{COL} * k)$
5: **else**
6: $N_{COL} \leftarrow N/nprocs$
7: C_{LOC} points to $C_{TEMP\_N} + (N_{COL} * k + rest)$
8: **end if**

9: **copy** the k block of B_{LOC} in B_{TEMP}
10: **calculate** the *receivecounts* and *displacements* for the *AllGatherv* operation

11: *AllGatherv(send:*B_{TEMP}*, receive:*B_{TEMP_N}*)*

12: $C_{LOC} \leftarrow matmul(A_{LOC}, B_{TEMP\_N})$

13: **end for**

14: **save** C_{TEMP_N} in binary file with MPI I/O

This Algorithm is performed by each process. The *matmul* function on CPU has two variants:

- *naive*: use the classical naive algorithm of Matrix Multiplication with 3 for loops on CPU
- *blas*: use the CBLAS dgemm function (multithreaded) on CPU to perform the matrix multiplication

3.2 GPU version

The GPU version of the algorithm is very similar to the previous but this time on CPU we perform only the initialization. All the pieces of matrices needed for an iteration are sended then to the GPUs which perform the computation phase with the *cublas_dgemm* function. Moreover, this function reads by default the matrix in column major order. To avoid each time the transposition, we instead calculate the results as $B^T A^T = C^T$. In this way, we already have B^T and A^T. Finally, the result C^T (that is stored in column major order) can be transposed to C by simply read it in row major order. After all the iterations, each process send the C_{TEMP_N} block stored on GPU to the CPU and saves the result in parallel. This version is shown in the Algorithm 2.

Algorithm 2 Parallel Matrix Multiplication Iteration - GPU

1: **allocate** $cu\_A_{LOC}, cu\_B_{TEMP\_N}, cu\_C_{TEMP\_N}$ on GPU
2: **memcpy** A_{LOC} in cu_A_{LOC}
3: **for** $k \leftarrow 0$ to $nprocs$ **do**
4: **if** $k <$ rest **then**
5: $N_{COL} \leftarrow N/nprocs + 1$
6: cu_C_{LOC} points to $cu\_C_{TEMP\_N} + (N_{COL} * k)$
7: **else**
8: $N_{COL} \leftarrow N/nprocs$
9: cu_C_{LOC} points to $cu\_C_{TEMP\_N} + (N_{COL} * k + rest)$
10: **end if**

11: **copy** the k block of B_{LOC} in B_{TEMP}
12: **calculate** the *receivecounts* and *displacements* for the *AllGatherv* operation

13: *AllGatherv(send:*B_{TEMP}*, receive:*$cu\_B_{TEMP\_N}$*)*

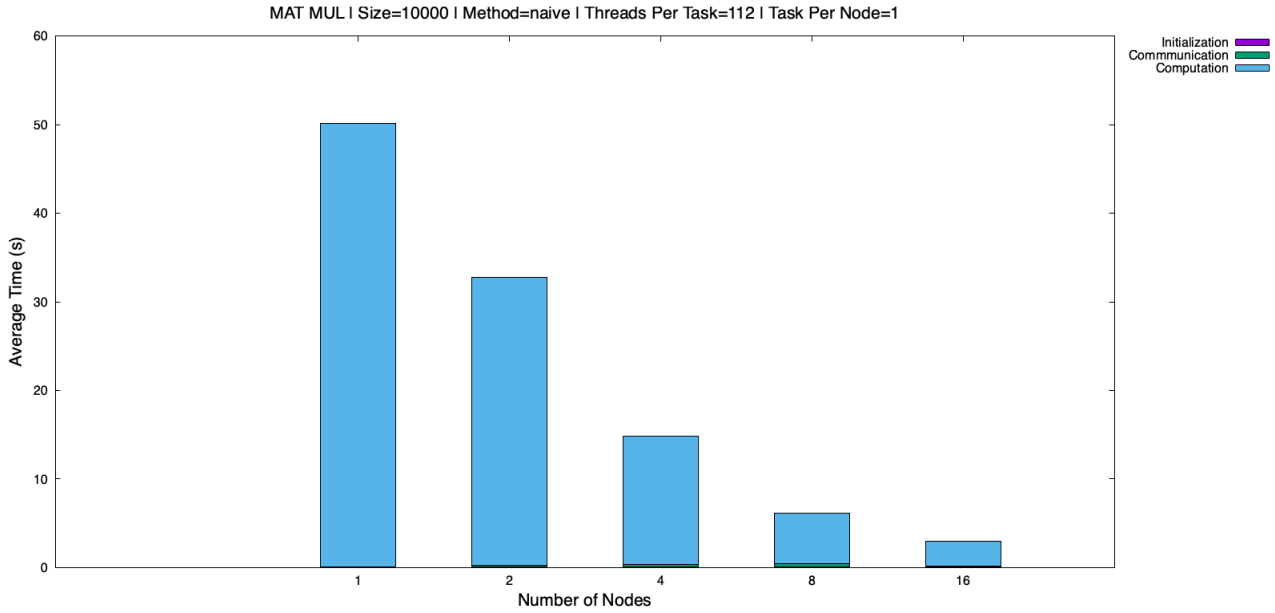
14: **memcpy** B_{TEMP_N} in $cu\_B_{TEMP\_N}$
15: $cu\_C_{LOC} \leftarrow cublas(cu\_B_{TEMP\_N}, cu\_A_{LOC})$

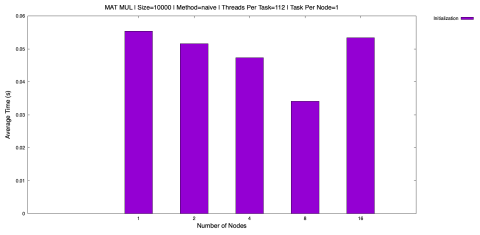
16: **end for**

17: **memcpy** $cu\_C_{TEMP\_N}$ in C_{TEMP_N}
18: **save** C_{TEMP_N} in binary file with MPI I/O

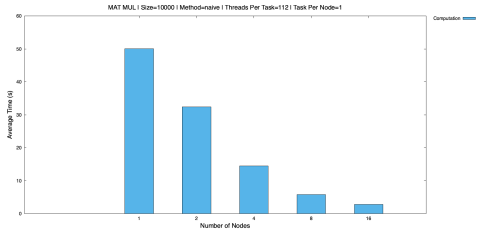
4 Naive Parallel Matrix Multiplication

4.1 Results - Size: 10000×10000

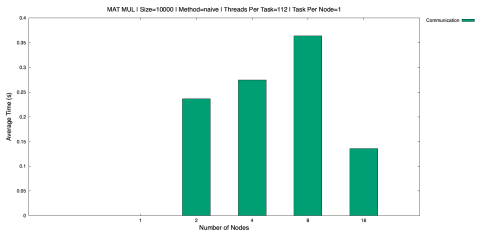




(a) Initialization



(b) Computation



(c) Communication

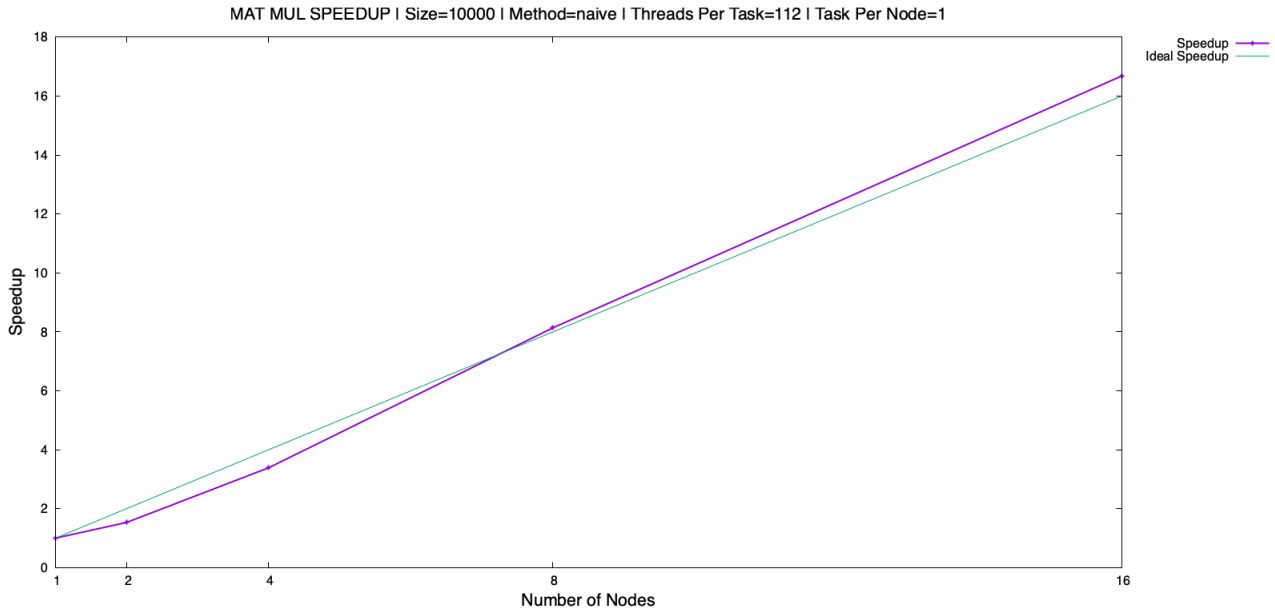
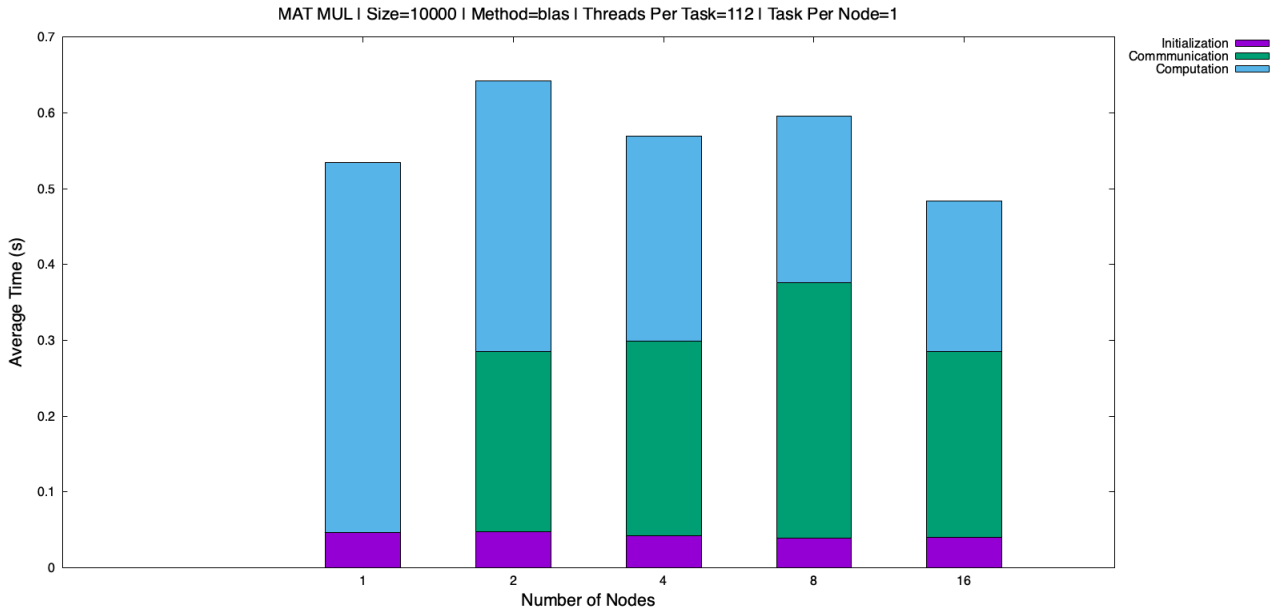
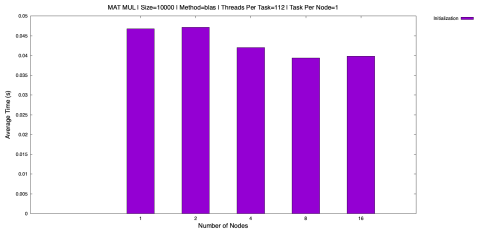


Figure 3: Speedup

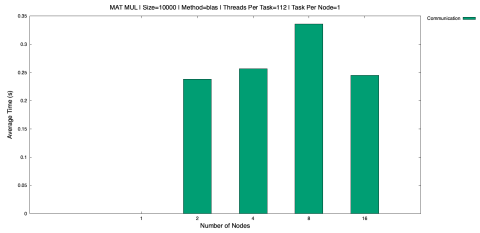
5 BLAS Parallel Matrix Multiplication

5.1 Results - Size: 10000×10000

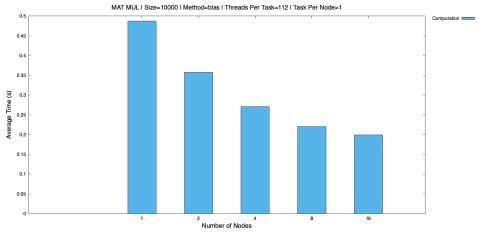




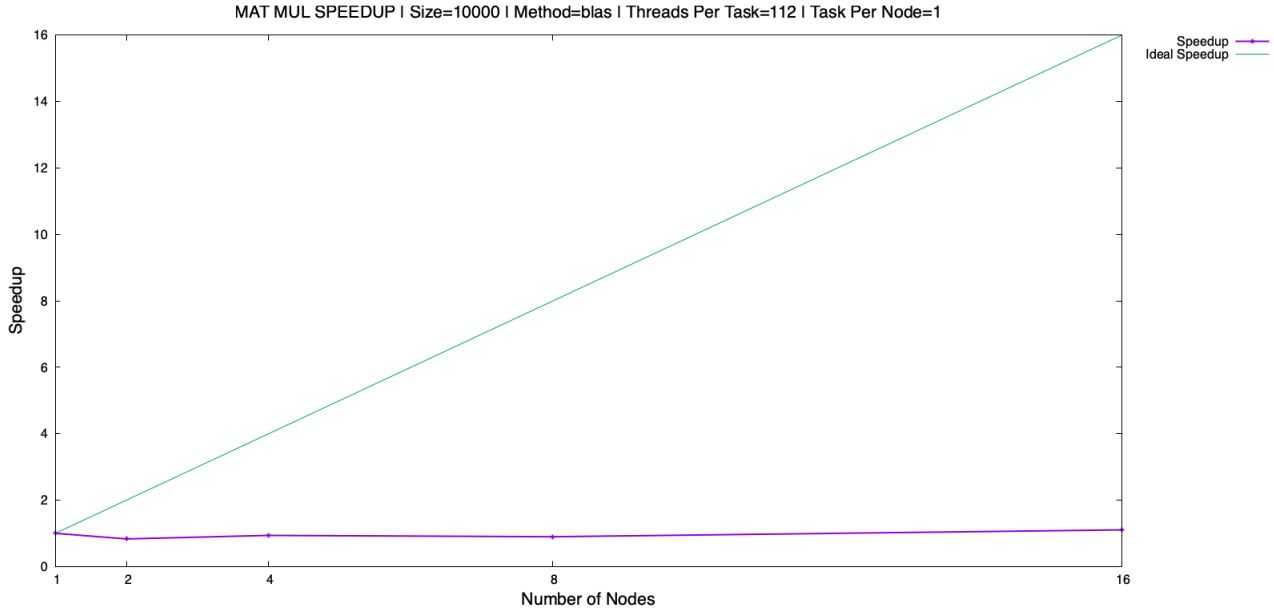
(a) Initialization



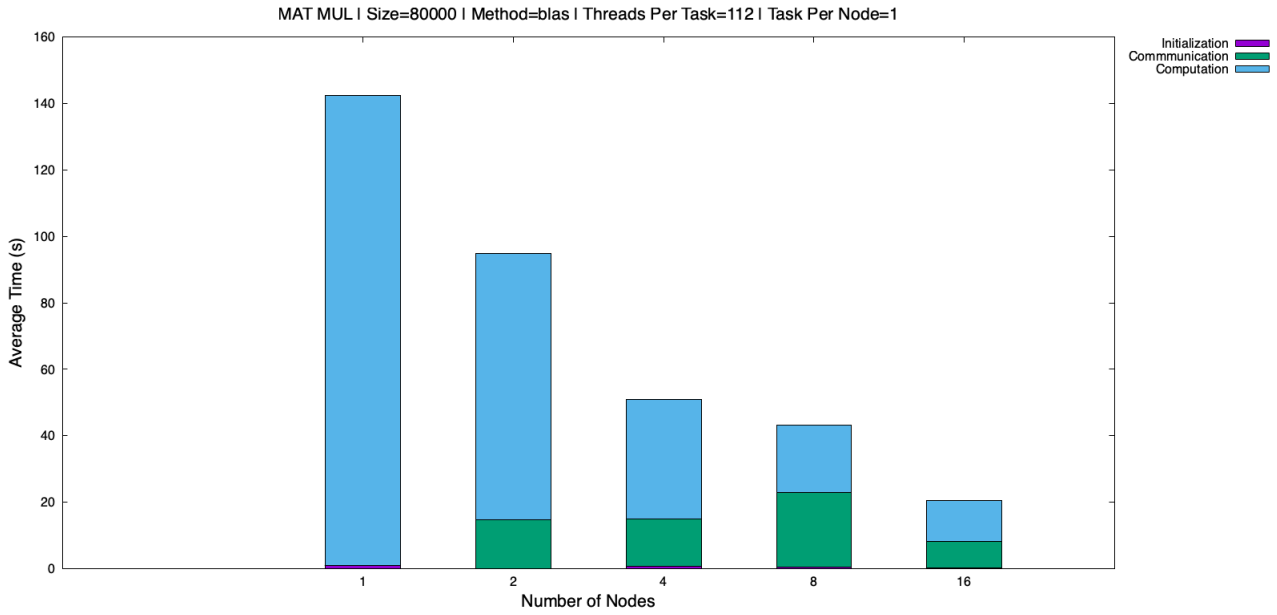
(b) Communication

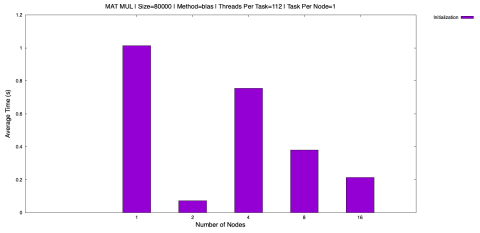


(c) Computation

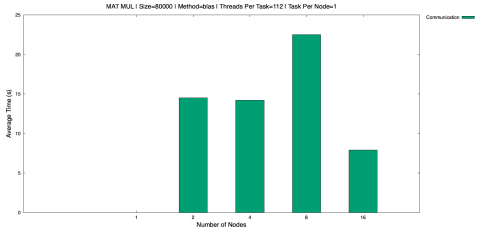


5.2 Results - Size: 80000×80000

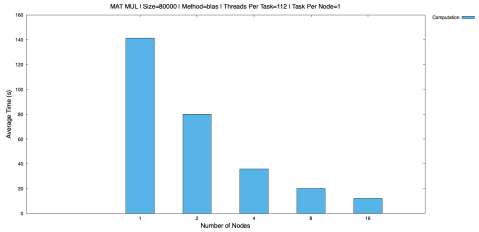




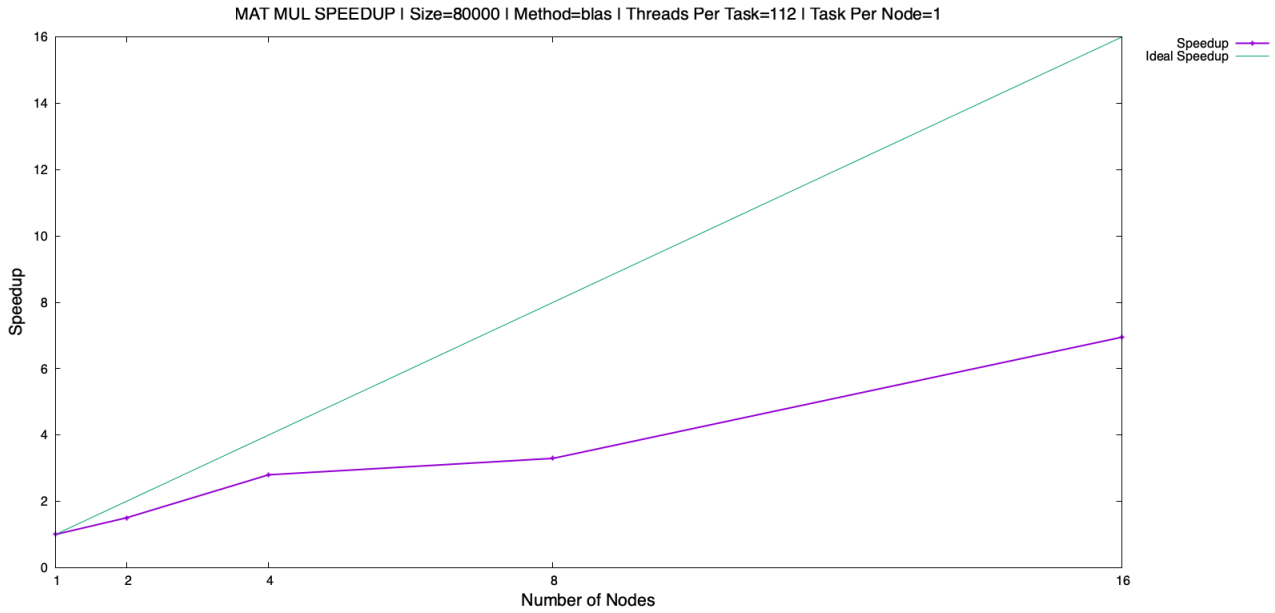
(a) Initialization



(b) Communication

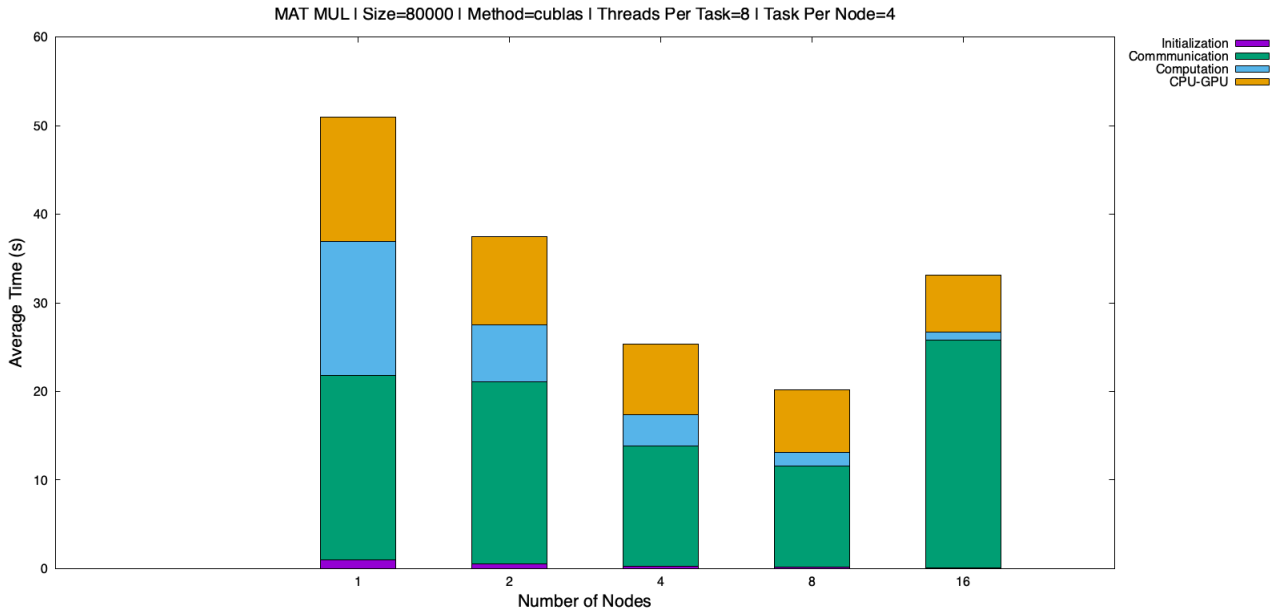


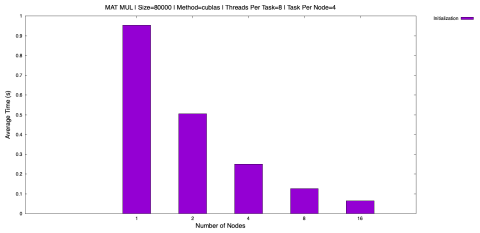
(c) Computation



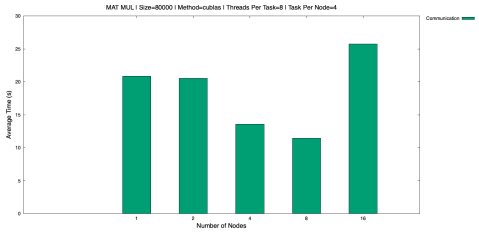
6 CUBLAS Parallel Matrix Multiplication

6.1 Results - Size: 80000×80000

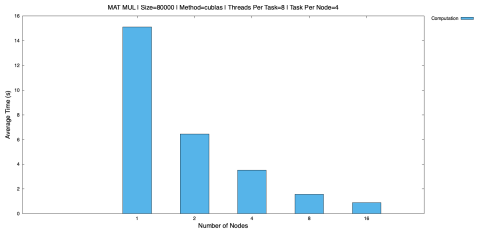




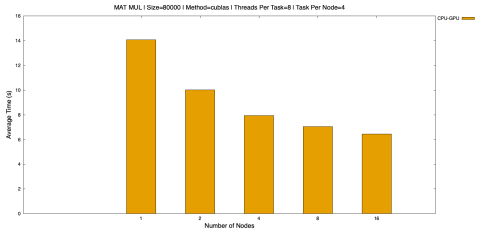
(a) Initialization



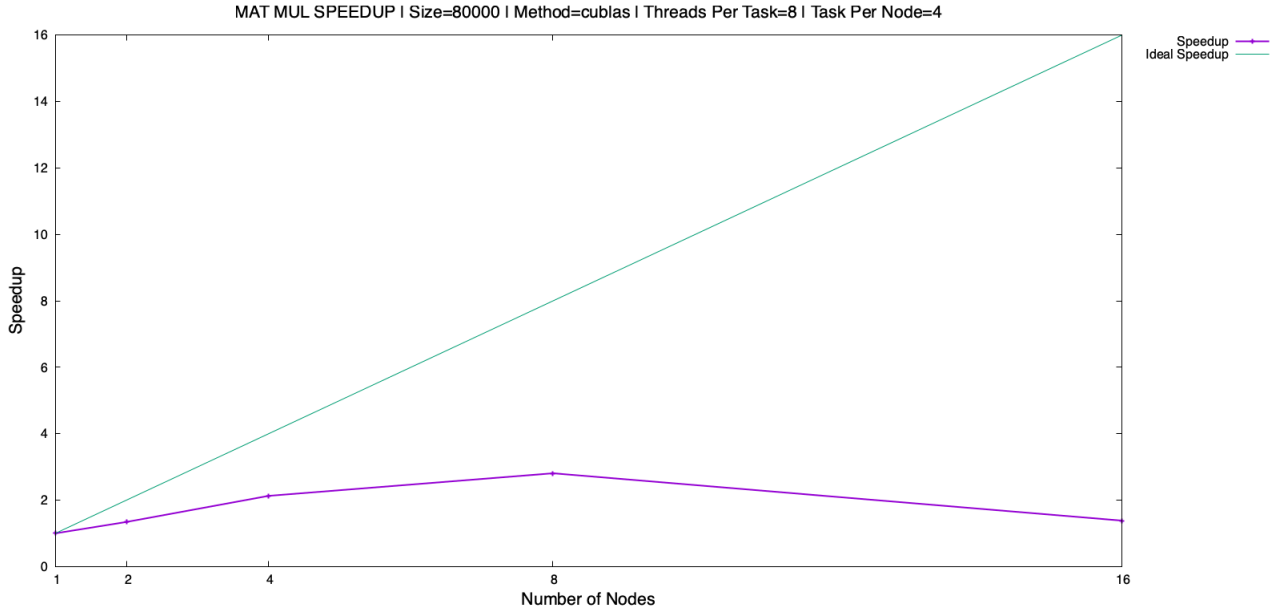
(b) Communication



(c) Computation



(d) CPU - GPU



7 Comparing Results

Table 1: MatMul BLAS vs CUBLAS - Size: 10000 × 10000

Number of Nodes	**1**	**2**	**4**	**8**	**16**
Initialization Time (s)					
blas	0.046762	0.047169	0.042010	0.039395	0.039865
naive	0.055397	0.051650	0.047349	0.034085	0.053422
Communication Time (s)					
blas	0.000000	0.237954	0.256770	0.336044	0.244944
naive	0.000000	0.236362	0.274776	0.364067	0.135636
Computation Time (s)					
blas	**0.487370**	**0.357325**	**0.270883**	**0.220210**	**0.198997**
naive	50.084395	32.435776	14.475794	5.762652	2.816488
Total Time (s)					
blas	**0.534132**	**0.642448**	**0.569663**	**0.595649**	**0.483806**
naive	50.139792	32.723788	14.797919	6.160804	3.005546

Table 2: MatMul BLAS vs CUBLAS - Size: 80000 × 80000

Number of Nodes	**1**	**2**	**4**	**8**	**16**
Initialization Time (s)					
blas	1.013511	**0.072719**	0.755066	0.380928	0.212851
cublas	**0.953505**	0.505618	**0.250112**	**0.126880**	**0.064923**
Communication Time (s)					
blas	0.000000	**14.526843**	14.203894	22.508671	**7.906259**
cublas	20.862211	20.564119	**13.570731**	**11.461551**	25.754809
Computation Time (s)					
blas	141.395141	80.197248	35.916465	20.336584	12.376804
cublas	**15.122763**	**6.443149**	**3.532343**	**1.565885**	**0.894362**
CPU-GPU Time (s)					
cublas	14.070837	10.008605	7.946371	7.042651	6.451190
Total Time (s)					
blas	142.408652	94.796810	50.875425	43.226183	**20.495914**
cublas	**51.009316**	**37.521491**	**25.299557**	**20.196967**	**33.165284**

Part II

Parallel Jacobi Iteration

8 CPU Two-Side MPI Jacobi Iteration

In this section we explain the algorithm to perform in parallel the Jacobi Iteration Method on a 2D grid using classical MPI non-blocking communications. We start by two matrix MAT and $MAT2$ of size $N \times N$. Each process allocates its own portion of these two matrix MAT_{LOC} and $MAT2_{LOC}$. Moreover, each internal process allocates two rows more (the ghost regions which will be sended), while the first and the last process allocate only one row (this because the first row and last row of the total matrix will not be shared). The schema of allocation is provided in Figure 7.

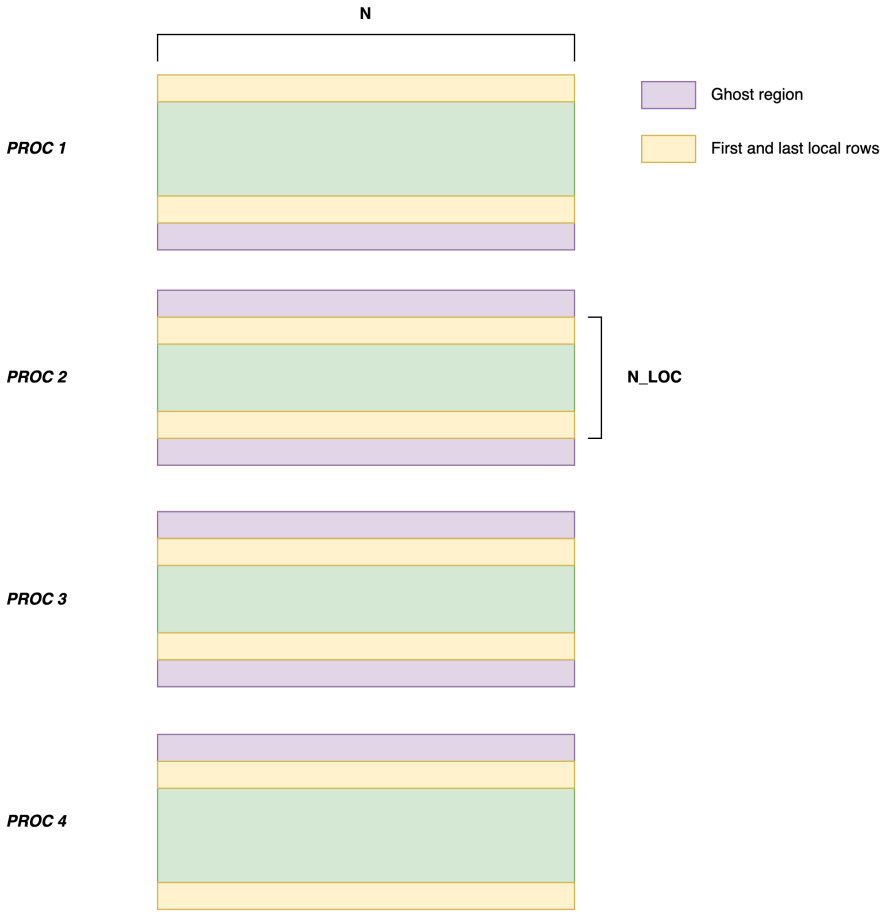


Figure 7: Jacobi Matrix Allocation for each process.

Next, each of 2 matrices is initialized (except for the ghost regions) in the following way:

- Top row and most right column of the matrix with all zeros
- Internal matrix values equal to 0.5

- Bottom row the most left column with a linear gradient from 0 to 100

An example of initialization is shown in Figure 8.

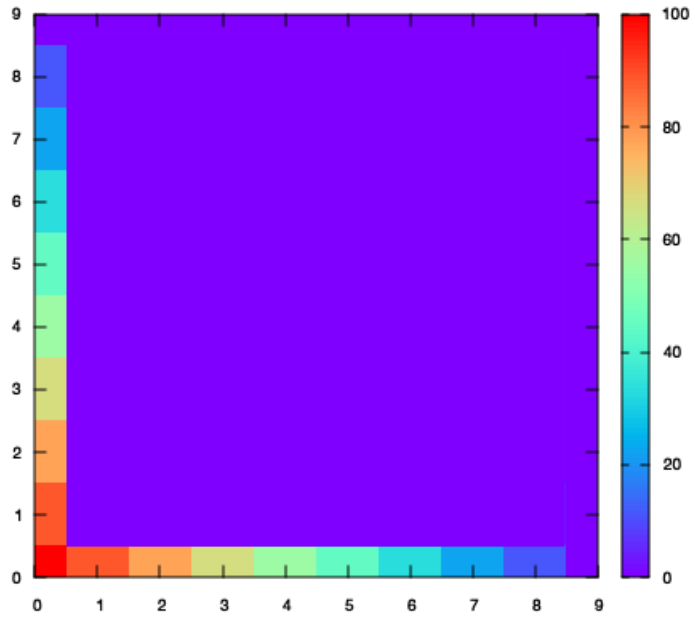


Figure 8: Initialization of a 10x10 matrix for the Jacobi Iteration.

After the initialization, each process starts the loop for the jacobi iteration. This last procedure is described in the Algorithm 3 and the communication pattern is shown in Figure 9. The first and last process do not send their first and last row.

Algorithm 3 Parallel Jacobi Iteration (Two Side Communication)

for $k \leftarrow 0$ to num_iter **do**
  $ghost\_up \leftarrow MAT_{LOC}$ ghost up region
  $ghost\_down \leftarrow MAT_{LOC}$ ghost down region
  $first\_row \leftarrow MAT_{LOC}$ first row
  $last\_row \leftarrow MAT_{LOC}$ last row
  if $proc \neq 0$ **then**
    $Isend(send{:}first\_row,\ to{:}proc-1)$
    $IRecv(receive\ in{:}ghost\_up,\ from{:}proc-1)$
  end if
  if $proc \neq nprocs-1$ **then**
    $Isend(send{:}last\_row,\ to{:}proc+1)$
    $Irecv(receive\ in{:}ghost\_down,\ from{:}proc+1)$
  end if
  $MPI\_Wait()$
  $MAT2_{LOC} \leftarrow JacobiUpdate(MAT_{LOC})$
  swap MAT_{LOC} *and* $MAT2_{LOC}$ *pointers*
end for

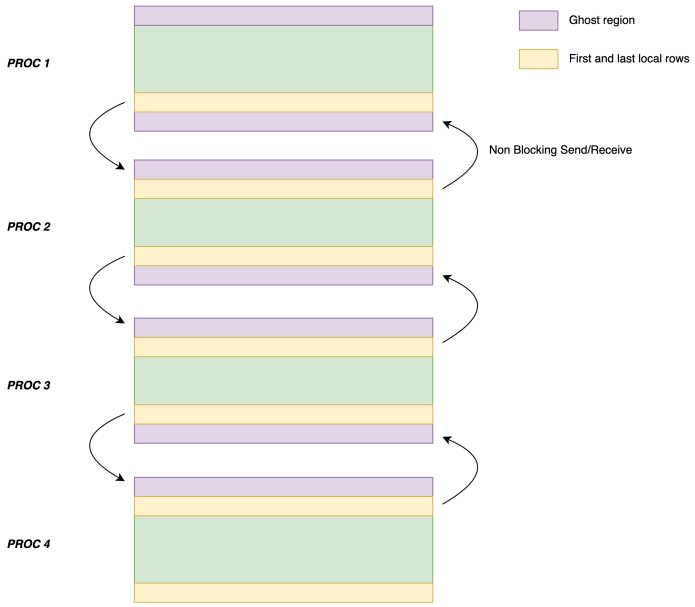


Figure 9: Parallel Jacobi Iteration Communication Pattern (Send/Receive).

When the loop finish, we save the matrix as a binary file in parallel using MPI I/O. An example plot of the final matrix is shown in Figure 10.

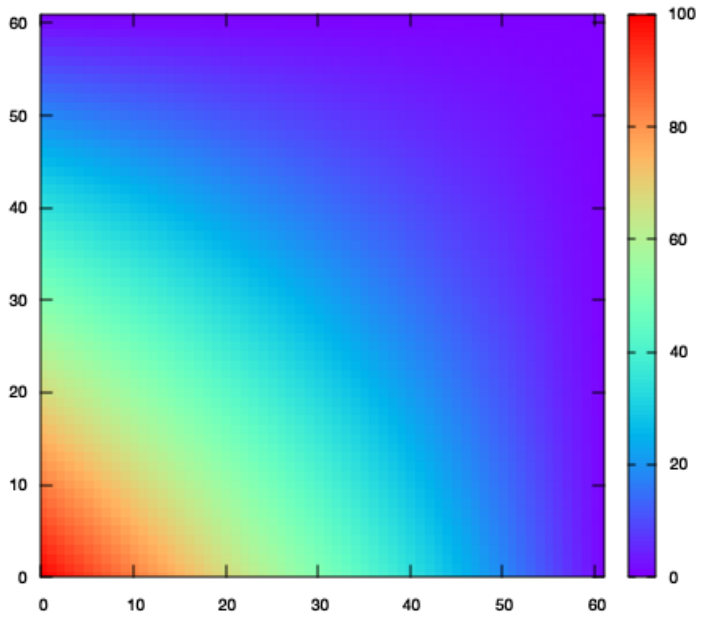
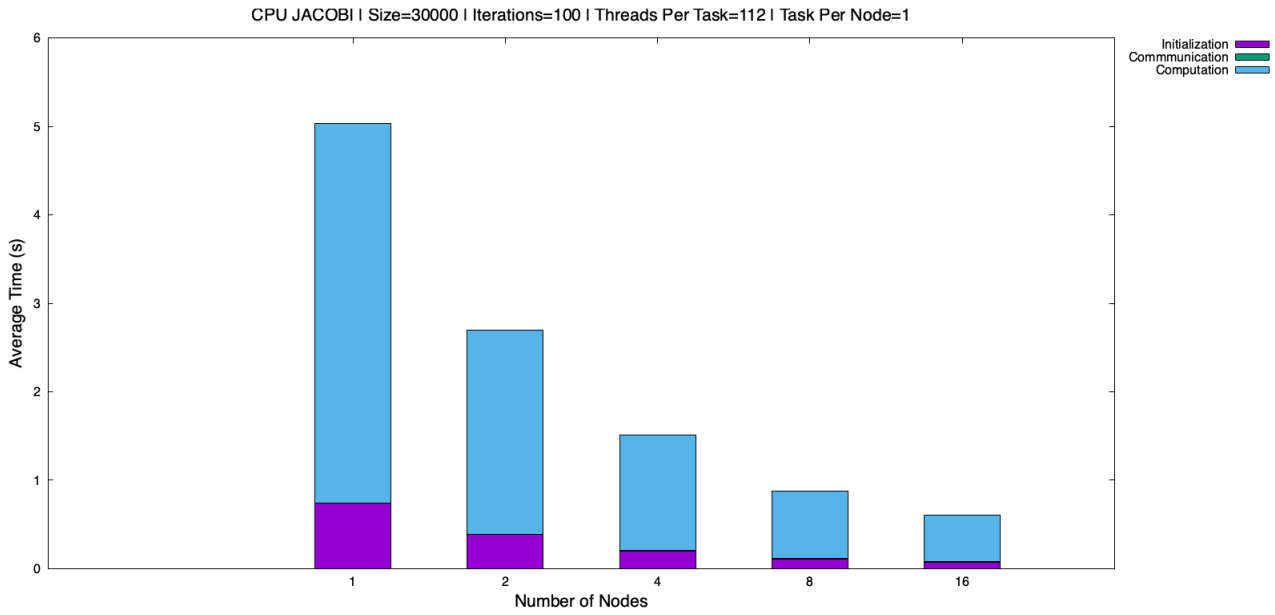
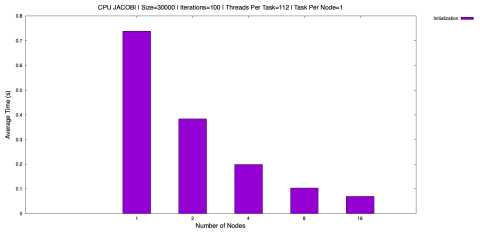


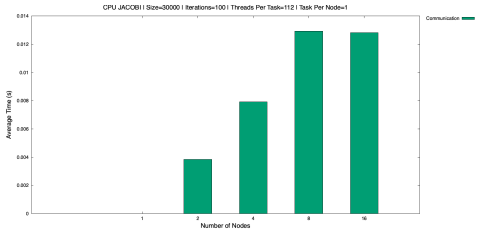
Figure 10: Final 60x60 matrix after 2000 Jacobi iterations.

8.1 Results - Size: 30000×30000, Iterations: 100

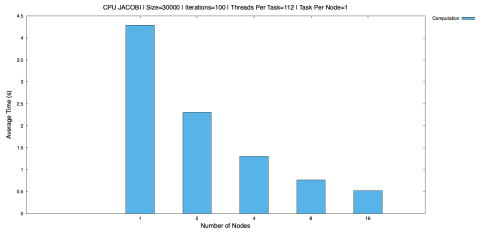




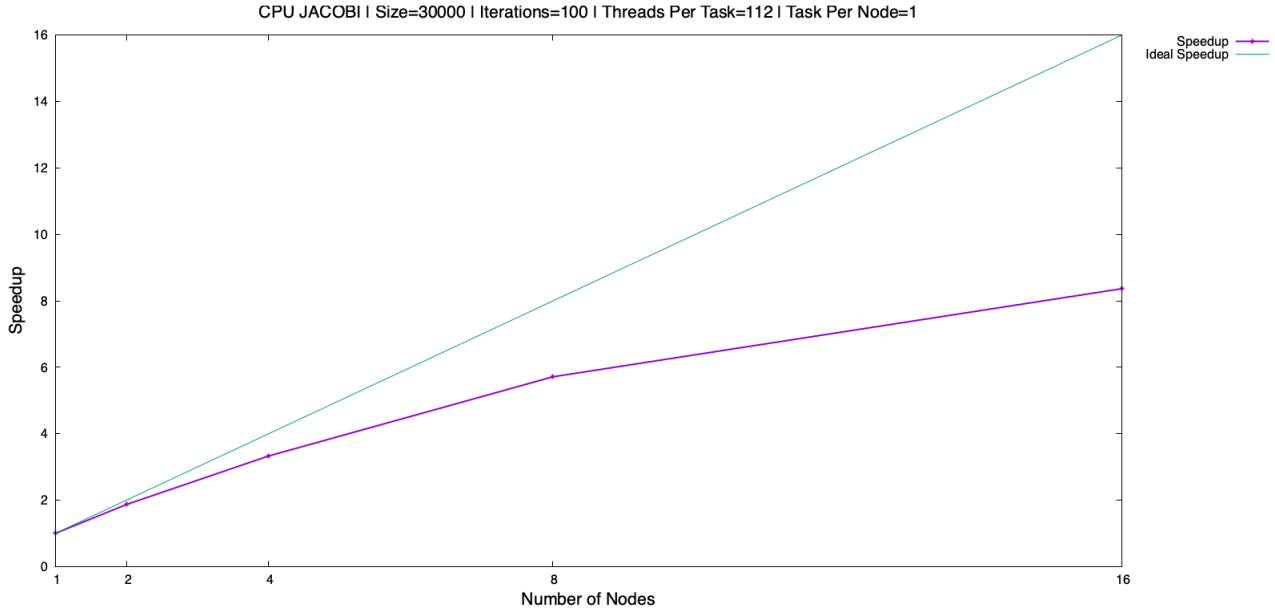
(a) Initialization



(b) Communication



(c) Computation



9 GPU Two-Side MPI (Cuda Aware) Jacobi Iteration with OpenACC

This implementation is equivalent to the previous one but this time we perform initialization, communication and computation all on GPUs using OpenACC and Cuda Aware MPI.

In particular, we call at the beginning of our code the directive *#pragma acc enter data create* to allocate MAT_{LOC} and $MAT2_{LOC}$ directly on GPU for each process. Then we perform all the initializations on GPU using *#pragma acc parallel loop present*$(MAT_{LOC}, MAT2_{LOC})$. For the communication, we wrap the MPI `ISend/IRecv` calls in the scope of the *#pragma acc host_data use_device*$(MAT_{LOC}, MAT2_{LOC})$, which enables us to expose the device pointers

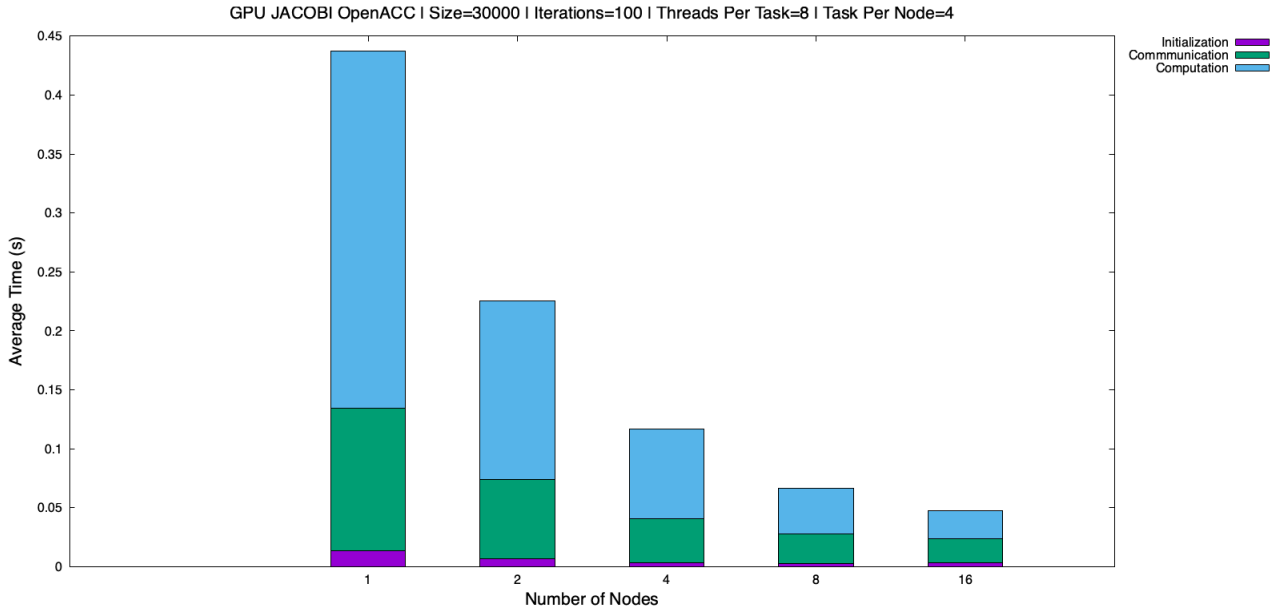
to the host and perform the MPI calls directly with them.

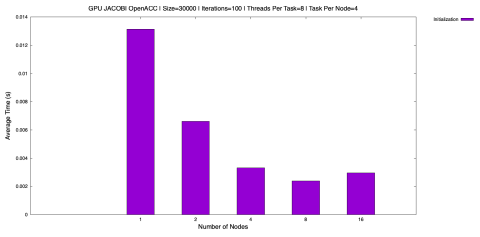
Then, the computation part is performed using again the *#pragma acc parallel loop present($MAT_{LOC}, MAT2_{LOC}$)* directive with the addition of the *#pragma acc loop independent* directive to improve the performance.

Finally, the I/O part is performed directly by the GPUs again using *#pragma acc host_data use_device($MAT2_{LOC}$)*.

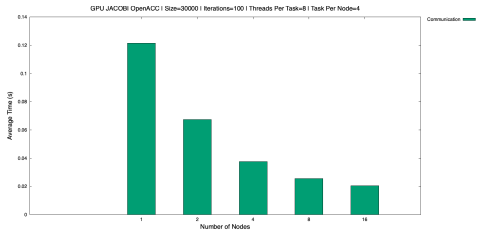
At the end we close the unstructured data region calling *#pragma acc exit data delete* directive on MAT_{LOC} and $MAT2_{LOC}$

9.1 Results - Size 30000×30000, Iterations: 100

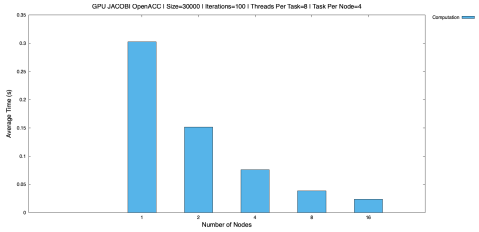




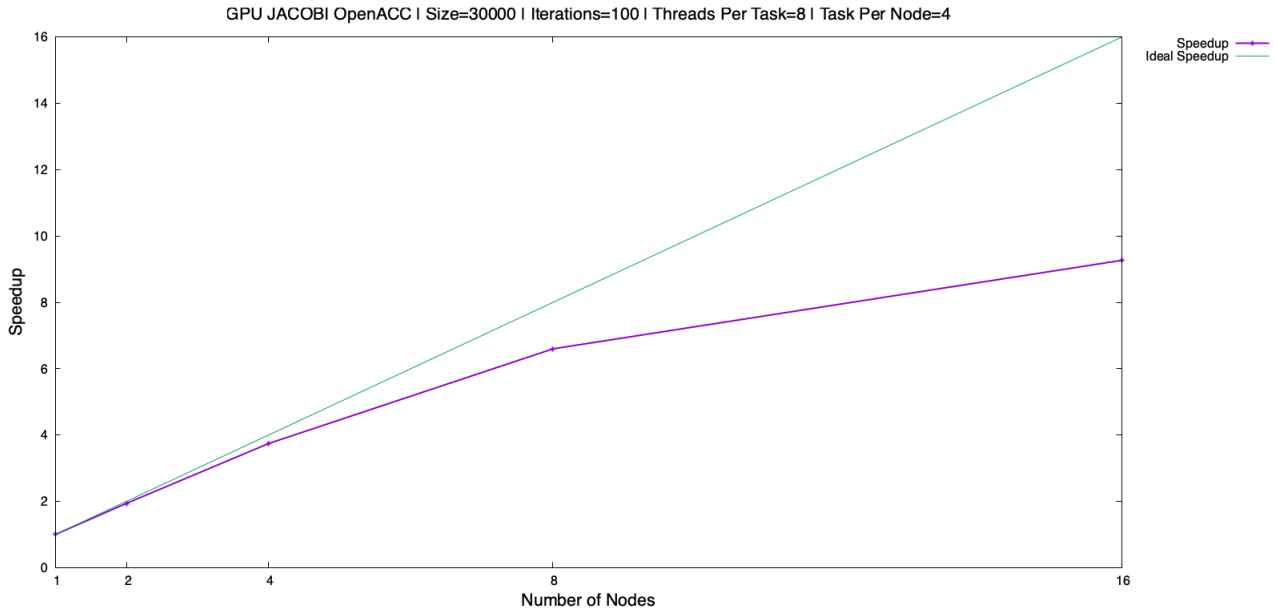
(a) Initialization



(b) Communication



(c) Computation



10 CPU One Side MPI Jacobi Iteration

In this section we explain the algorithm to perform in parallel the Jacobi Iteration Method on a 2D grid using One ide MPI communications.

There are two sub-implementations: the first use the *MPI_Put* function while the second the *MPI_Get*. We start again with two matrix MAT and $MAT2$ and using the same allocation and initialization of the Two-side version.

Then, we calculate the pointers for the first local row, last local row and the two ghost region for each of the 2 matrices.

Next, we define and create the two memory windows that will be shared. In the case of *MPI_Put* version the windows points to the ghost regions for each of the two matrices Instead, with the *MPI_Get* routine, the windows will point to the first and to the last row of each matrices. So, in general, we have 2 windows per matrix.

Moreover, we optimize the windows creation by setting the related *MPI_Info* with the following attributes equal to true:

- *same_size*: suggests to the compiler that size is identical on all processes
- *same_disp_unit*: suggest to the compiler that the unit displacement is identical on all processes

Finally, we start with the Jacobi iteration loop in which each process uses the chosen routine. We illustrate the two versions in the Algorithms 4 and 5. Both will be syncronized using the *MPI_Win_lock MPI_Win_unlock* calls and placing an *MPI Barrier* after. Moreover, we use the lock calls with two argument:

- *MPI_LOCK_EXCLUSIVE*: ensure that no other process can access the window concurrently.
- *MPI_MODE_NOCHECK*: optimization that skips the lock checking mechanism to save time.

At each iteration we swap the pointers of the matrices and change the windows used for communication: in this way we can create only two windows at the beginning and not at each iteration. The working of this method is illustrated in the Figure 13.

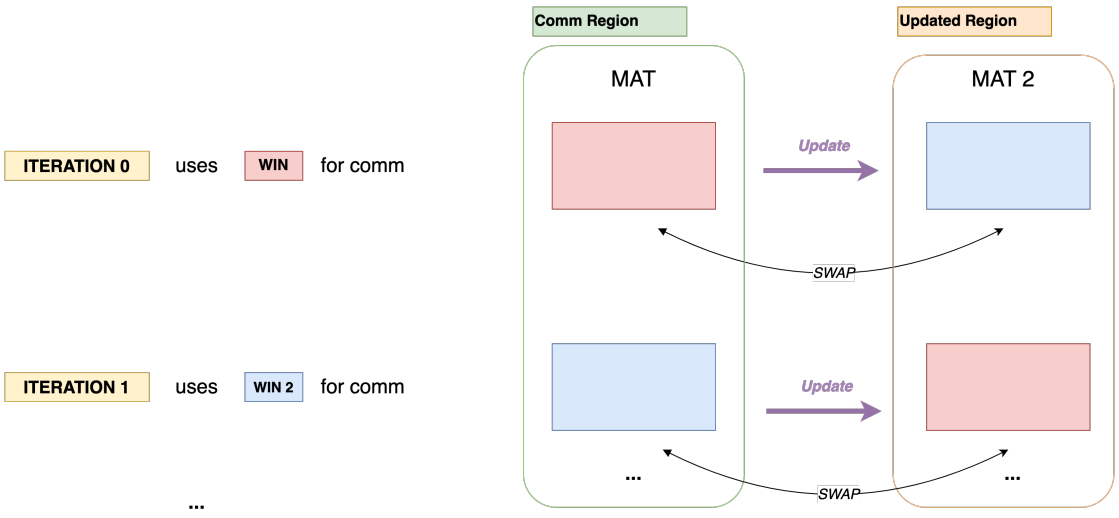


Figure 13: Iterations of the One side Jacobi Algorithm

Algorithm 4 Parallel Jacobi Iteration (One Side PUT Communication)

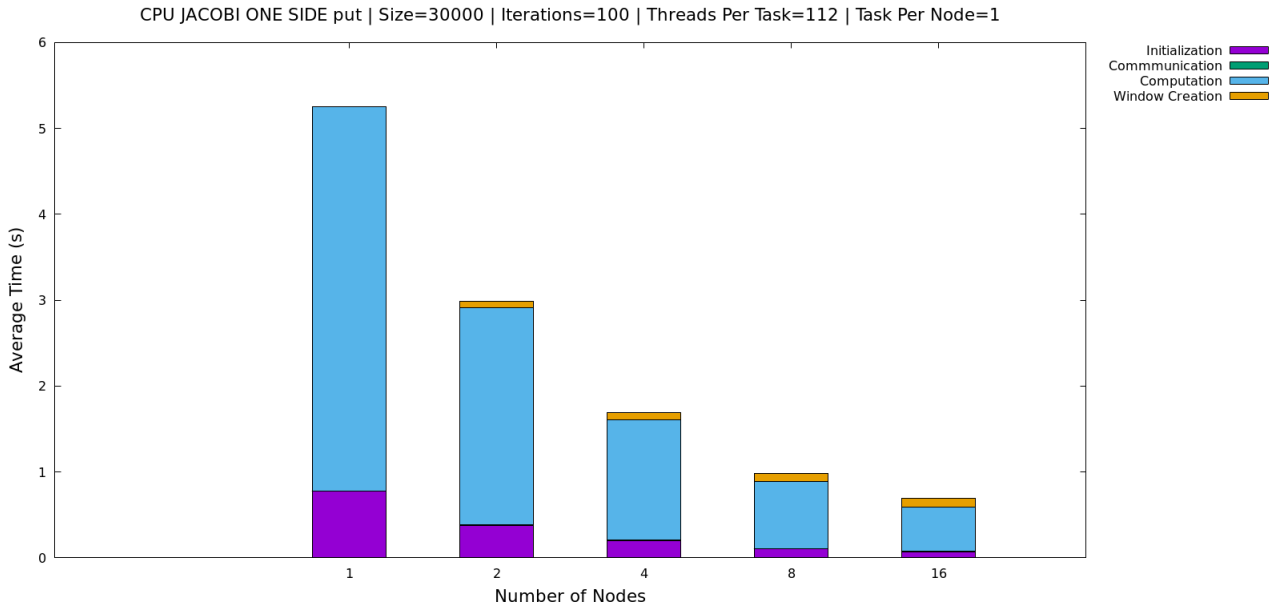
1: $ghost\_up\_win \leftarrow MAT_{LOC}$ ghost up region window
2: $ghost\_down\_win \leftarrow MAT_{LOC}$ ghost down region window
3: $ghost\_up\_win\_2 \leftarrow MAT2_{LOC}$ ghost up region window
4: $ghost\_down\_win\_2 \leftarrow MAT2_{LOC}$ ghost down region window
5: **for** $k \leftarrow 0$ to num_iter **do**
6: **if** $k\%2 = 0$ **then**
7: **if** $proc \neq 0$ **then**
8: *WinLock(ghost_down_win)*
9: $PUT(send: first\_row,\ to: proc - 1,\ in: ghost\_down\_win)$
10: *WinUnlock(ghost_down_win)*
11: **end if**
12: **if** $proc \neq nprocs - 1$ **then**
13: *WinLock(ghost_up_win)*
14: $PUT(send: last\_row,\ to: proc + 1,\ in: ghost\_up\_win)$
15: *WinUnlock(ghost_up_win)*
16: **end if**
17: **else**
18: Same as before but using $MAT2\_LOC$ windows
19: **end if**
20: *MPI_Barrier()*
21: $MAT2_{LOC} \leftarrow JacobiUpdate(MAT_{LOC})$
22: **swap** MAT_{LOC} *and* $MAT2_{LOC}$ *pointers*
23: **end for**

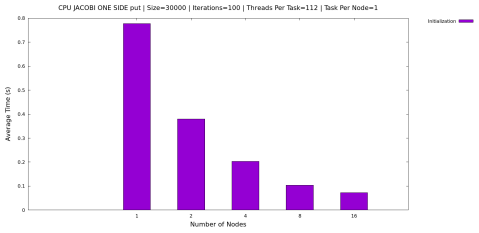
Again, When the loop finish we save the matrix as a binary file in parallel using MPI I/O.

Algorithm 5 Parallel Jacobi Iteration (One Side GET Communication)

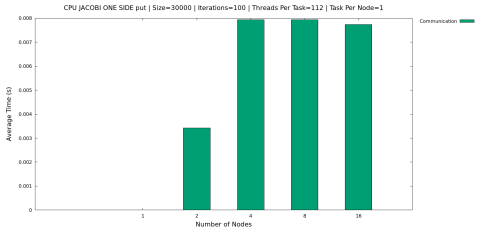
1: $first\_row\_win \leftarrow MAT_{LOC}$ first row window
2: $last\_row\_win \leftarrow MAT_{LOC}$ last row window
3: $first\_row\_win\_2 \leftarrow MAT2_{LOC}$ first row window
4: $last\_row\_win\_2 \leftarrow MAT2_{LOC}$ last row window
5: **for** $k \leftarrow 0$ to num_iter **do**
6: **if** $k\%2 = 0$ **then**
7: **if** $proc \neq 0$ **then**
8: *WinLock(last_row_win)*
9: *GET(from:* $last\_row\_win$*, of:* $proc - 1$*, in:ghost_up_win)*
10: *WinUnlock(last_row_win)*
11: **end if**
12: **if** $proc \neq nprocs - 1$ **then**
13: *WinLock(first_row_win)*
14: *GET(from:* $first\_row\_win$*, of:* $proc + 1$*, in:ghost_down_win)*
15: *WinUnlock(first_row_win)*
16: **end if**
17: **else**
18: Same as before but using $MAT2\_LOC$ windows
19: **end if**
20: *MPI_Barrier()*
21: $MAT2_{LOC} \leftarrow JacobiUpdate(MAT_{LOC})$
22: **swap** MAT_{LOC} *and* $MAT2_{LOC}$ *pointers*
23: **end for**

10.1 Results - PUT, Size 30000×30000, Iterations: 100

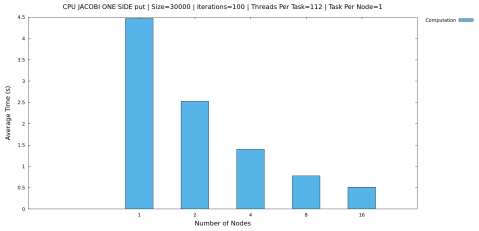




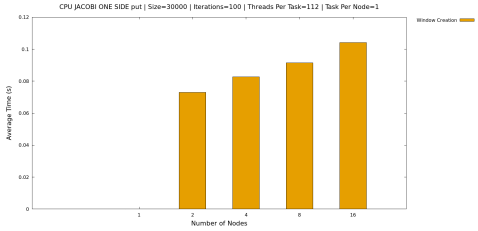
(a) Initialization



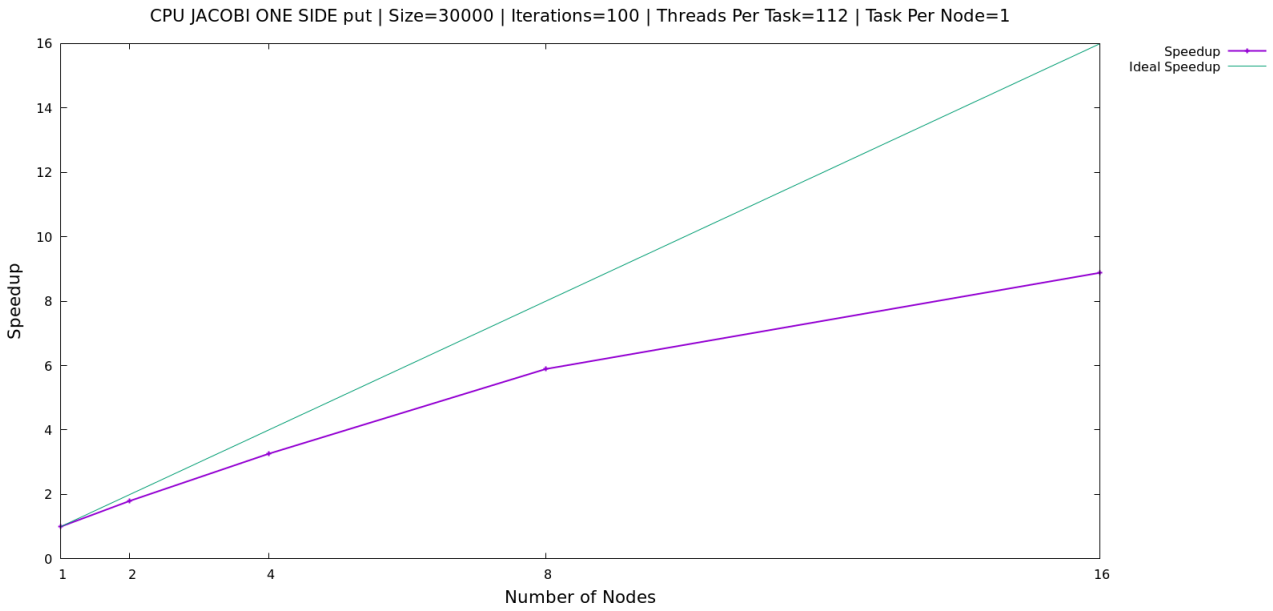
(b) Communication



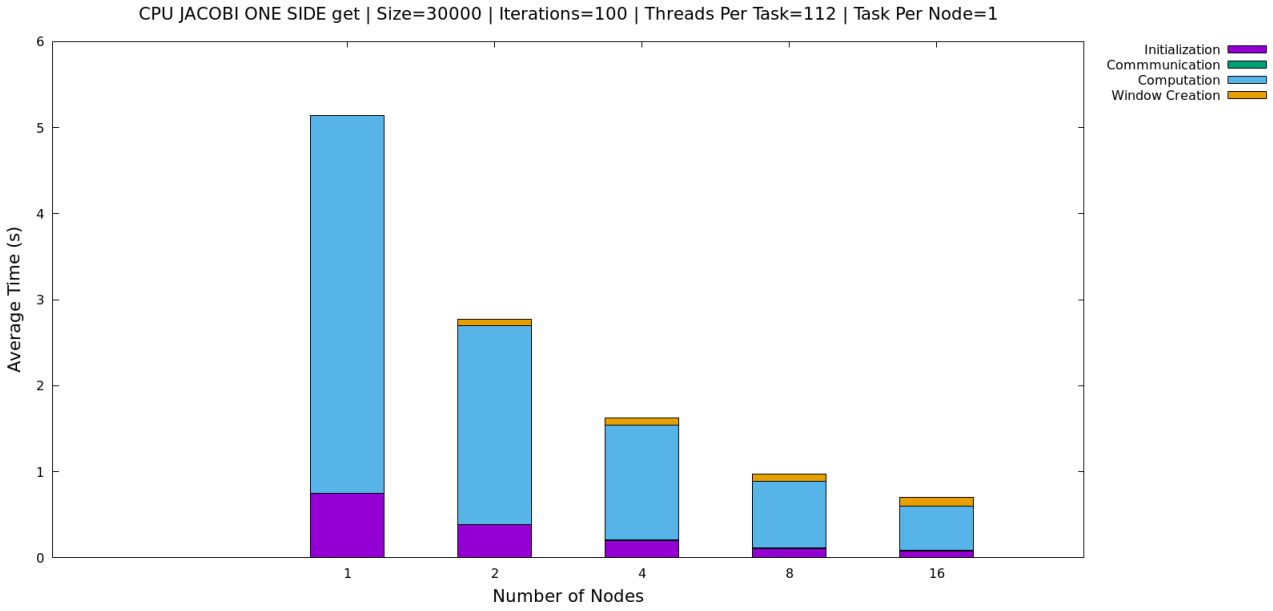
(c) Computation

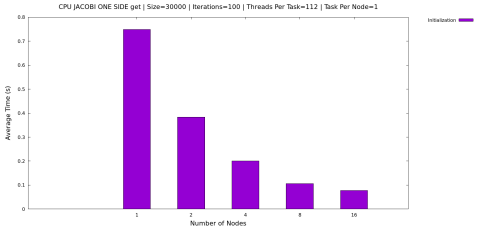


(d) Window Creation

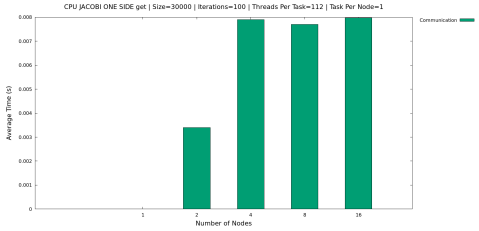


10.2 Results - GET, Size 30000×30000, Iterations: 100

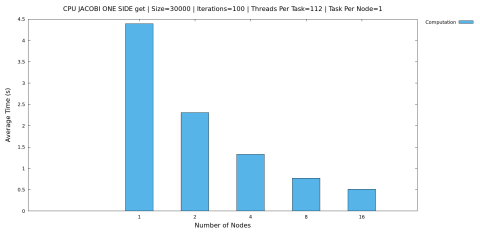




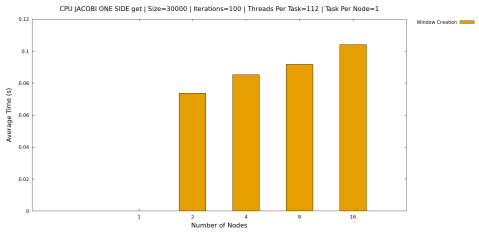
(a) Initialization



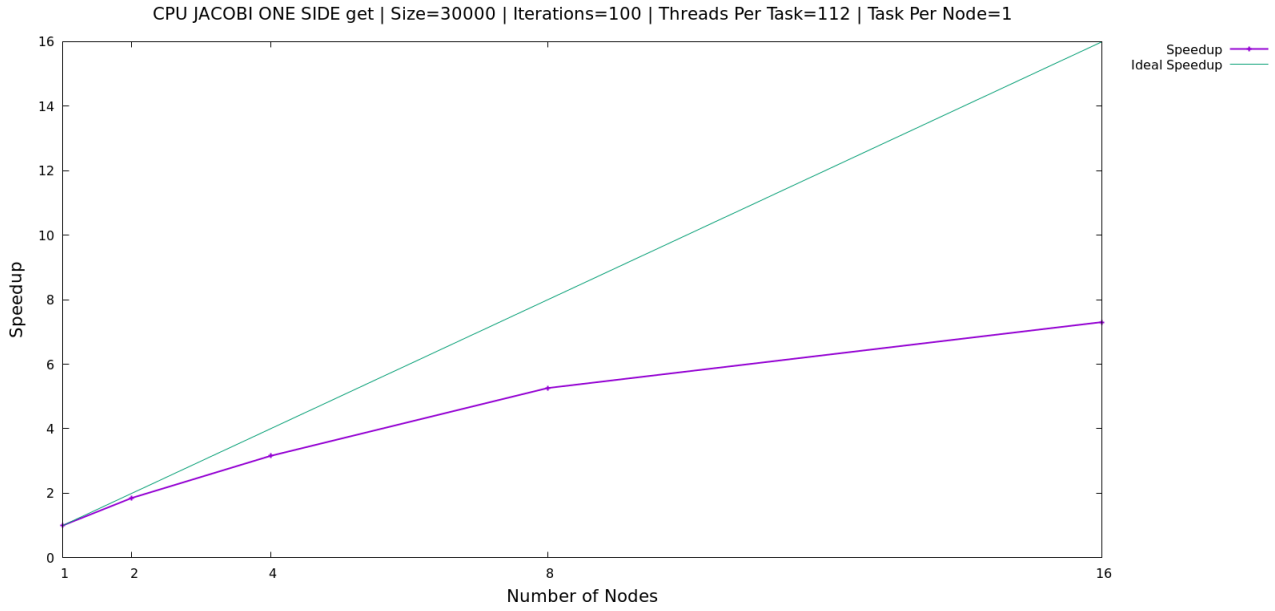
(b) Communication



(c) Computation



(d) Window Creation



11 Comparing Results

Table 3: Jacobi Two-CPU vs Two-GPU vs Get-CPU vs Put-CPU - Size: 80000 × 80000

Number of Nodes	**1**	**2**	**4**	**8**	**16**
Initialization Time (s)					
two	0.741305	0.383529	0.199044	0.107419	0.063435
acc	**0.013149**	**0.006618**	**0.003316**	**0.002394**	**0.002959**
get	0.748542	0.383122	0.201019	0.106712	0.077819
put	0.776928	0.380103	0.202724	0.103180	0.072892
Communication Time (s)					
two	0.000006	0.003741	**0.007838**	0.007772	0.008648
acc	0.121487	0.067450	0.037586	0.025486	0.020540
get	0.000000	**0.003393**	0.007894	**0.007697**	0.007971
put	0.000000	0.003429	0.007939	0.007931	**0.007737**
Computation Time (s)					
two	4.244624	2.223538	1.213914	0.865562	0.439050
acc	**0.302801**	**0.151623**	**0.075955**	**0.038409**	**0.023701**
get	4.391596	2.309921	1.333212	0.770572	0.513563
put	4.472803	2.530749	1.398870	0.779321	0.510175
Windows Creation Time (s)					
get	0.000000	0.073715	0.085325	0.091813	0.104076
put	0.000025	0.072929	0.082611	0.091378	0.104103
Total Time (s)					
two	4.985935	2.610808	1.420796	0.980753	0.511133
acc	**0.437437**	**0.225691**	**0.116857**	**0.066289**	**0.047200**
get	5.140138	2.770151	1.627450	0.976794	0.703429
put	5.249756	2.987210	1.692144	0.981810	0.694907