# ADV-HPC

**Matteo Nunziante**
Università degli studi di Trieste

June 2024

# Contents

# Part I

# Matrix Multiplication

## 1 Introduction

This part is about the performance analysis of matrix multiplication in a distributed OpenMP + MPI + CUDA setting. Matrix product it is level 3 Blas function, it perform $O(N^3)$ operations on $O(N^2)$ data.

- **Naive** Matrix Multiplication using parallel nested loops.
- **C-BLAS** using dgemm function.
- **CU-BLAS** using cublasDgemm function.

The chosen function will perform the usual matrix multiplication $C = A \cdot B$

## 2 Methodology

The goal is to implement multiple distributed matrix multiplication algorithms compatible with preexisting code. The main focus is on the performance of the algorithms in terms of execution time for the different sections as follows:

- **Initialization** Always perfromed on CPU. A is initialized to the Identity while elements of B follow an increasing sequence from 0 to N*N.
- **Communication** is performed to gather the columns of B.
- **Computation** performed using the above mentioned routines.

Analysis of performance is also based on the Speedup metric to measure the relative performance of the serial (1 process) and multiprocess implementations. For a given workload of k (for instance the dimension of the matrix) the speedup for *size* workers is defined as:

$$S_{size} = \frac{T_1}{T_{size}}$$

Note about timings: for each section execution time is the maximum of the times measured for each process.

## 3 Implementation

### 3.1 Initialization

Initialization is always performed on CPU using OMP threads, each process initializes a number of rows given by the formula:

$$N_{loc} = N/\text{size} + (\text{rank} < N\%size)$$

Where $N$ is the number or rows of the matrix (and columns, assuming it is a square matrix), *size* is the number of mpi processes and *rank* is the id of the process.

This is repeated for matrix $A$, $B$ and $C$.

```
double* sequence_initialize(unsigned long long int N,unsigned long long int M
    ,int size, int rank, char* fname)
{
    unsigned long long int N_loc=N, offset=0;

    if (size > 1){
        N_loc = N/size + 1*(rank < N%size);
        offset = rank*N_loc + N%size*(rank >= N%size);
    }

    double *A = (double*)malloc(N_loc*M*sizeof(double));

    #pragma omp parallel for
    for (int i = 0; i < N_loc*M; i++)
    {
            A[i] = (offset*M+i);
    }
    return;
}
```

## 3.2 Communication

By construction, matrix matrix product iterates on rows of $A$ and columns of $B$, this results in the need to gather the columns of B aggregating rows from every process. This is achieved with an All Gather.

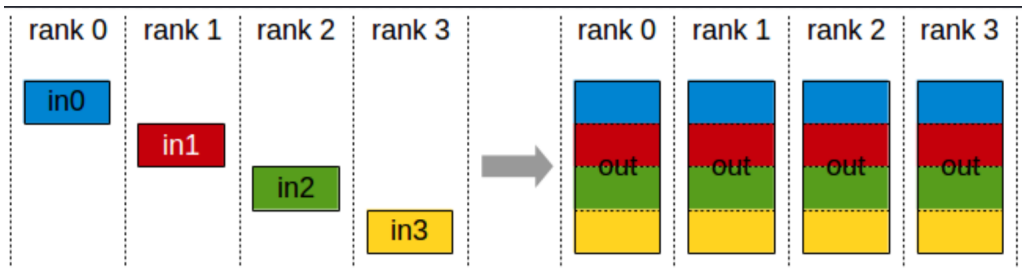



Figure 1: All Gather

Since it only works on contingous blocks of memory, each process has to copy the selected columns in a new array, it can than proceed to call the MPI_All_Gather_v function (all gather v is necessary since the number of elements sent depends on the rows assigned to that process).

```
void cpu_copy_gather(double* restrict B,double* restrict B_loc, double*
    restrict B_mult, unsigned long long int N_loc, unsigned long long int N,
    unsigned long long int ncol, int *nrows, int *displs)
{
    #pragma omp parallel for collapse(2)
    for(unsigned long long int k = 0; k < N_loc; k++){
        for(unsigned long long int j = 0; j < ncol; j++){
            B_loc[k*ncol+j] = B[ N*k + j ];
```

```
        }
    }

    MPI_Allgatherv(B_loc, N_loc*ncol, MPI_DOUBLE, B_mult, nrows, displs,
    MPI_DOUBLE, MPI_COMM_WORLD);

}
```

Matrix allocation is performed only once at the beginning of the program, since tests showed it is a quite consuming operation.

## 3.3 Computation

The actual product is performed block-wise among rows of matrix $A$ stored in the process memory and columns of matrix $B$ gathered as above. This is repeated *size* times as columns of $B$ are gathered in chuncks of number of columns $ncol = N/size + (i < N\%size)$ where $i$ is an index ranging from 0 to $size - 1$

**Algorithm 2** Matrix Multiplication

1: **for** $i = 1, 2, \ldots, size$ **do**
2: B_mult = Gather(B,i)
3: Copy B_mult to Device
4: mult(A,B_mult,C,i)
5: **end for**

Figure 2: Matrix Multiplication

### 3.3.1 GPU Computation

A quick note about the GPU computation.
Since the cuBLAS library uses column-major storage for matrices, the product is performed exploiting the fact that a row-major matrix is equivalent to the transposed in column-major order and the simple linear algebra property:

$$A \cdot B = C \iff B^T \cdot A^T = C^T$$

The kernel call is performed as follows:

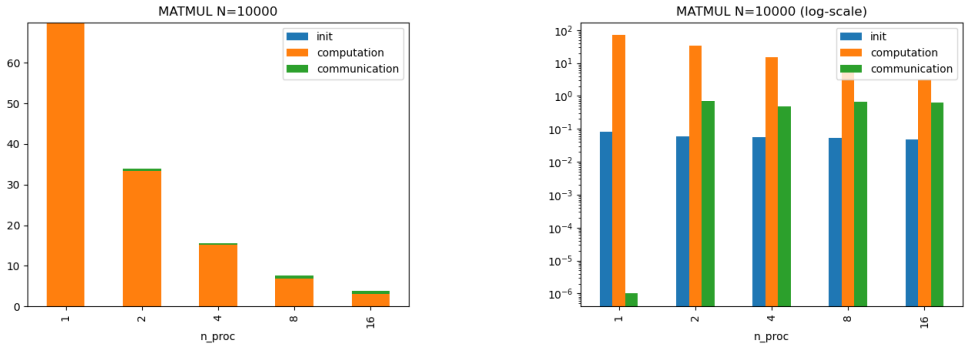
```
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, ncol, N_loc, N, &alpha,
    d_B_mult, ncol, d_A, N, &beta, &d_C[offset], N);
```

where *ncol* is the number of columns of B, $N$ is the number of rows of B and columns of A, *N_loc* is the number of rows of A. An offset C pointer is passed as usual to avoid the need of temporary C matrix and additional copies.

# 4 Scaling Results

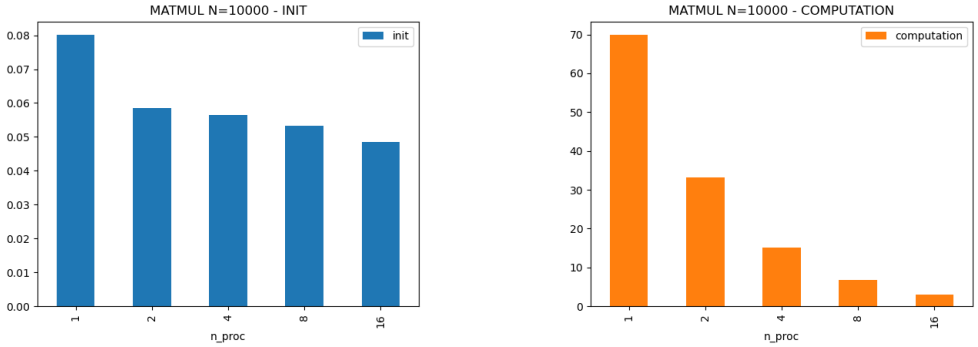
## 4.1 Matmul

Scaling is performed mapping processes to nodes (2x56 cores Intel Sapphire Rapids at 2.00 GHz) using all threads available (112 - no hyperthreading).




(a) label 1 (b) label 2

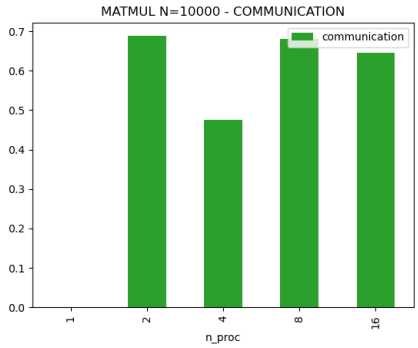
Figure 3: Matmul Scaling



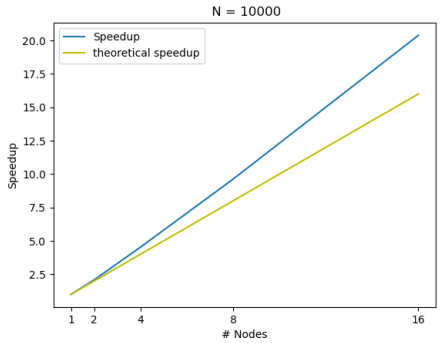

(a) label 1 (b) label 2

Figure 4: Matmul init and comp

- **Initialization** doesn't scale well with the number of processes, this is due to the fact that the size of the matrix doesn't justify the overhead of threads creation.




(a) label 1



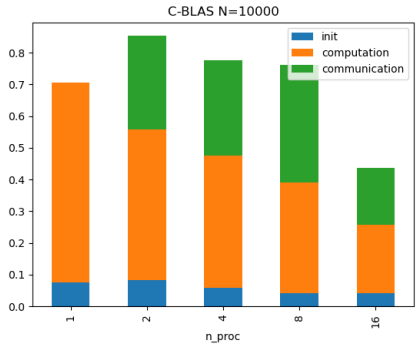

(b) label 2

Figure 5: Matmul comm and speedup

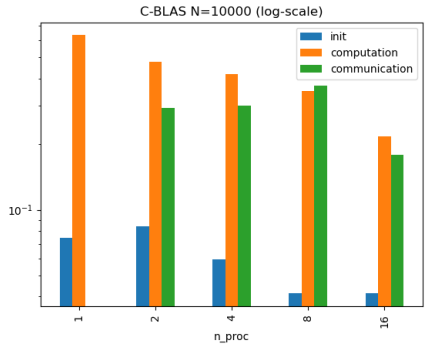
## 4.2 C-BLAS

Scaling is performed mapping processes to nodes (2x56 cores Intel Sapphire Rapids at 2.00 GHz) using all threads available (112 - no hyperthreading).

## 4.3 N = 10000

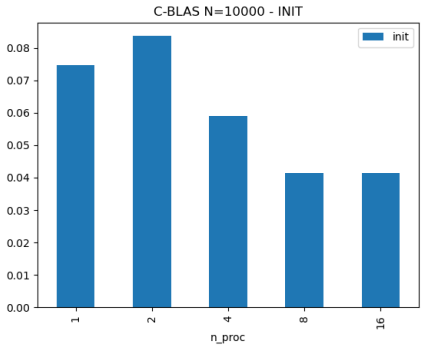



(a) label 1

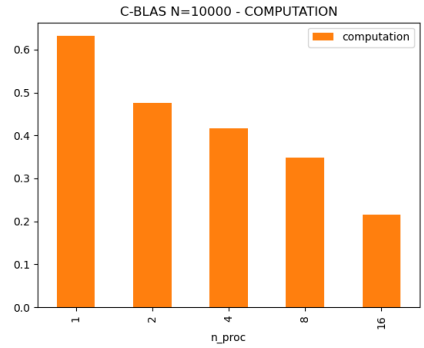



(b) label 2

Figure 6: C-BLAS Scaling

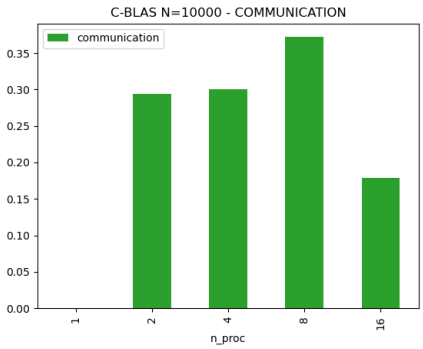



(a) label 1

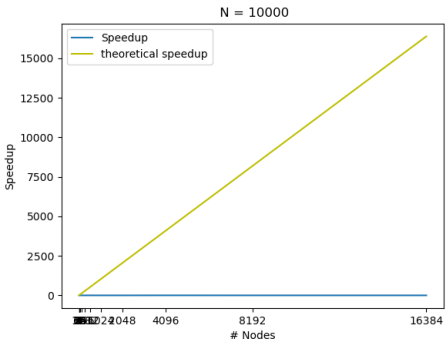



(b) label 2

Figure 7: C-BLAS init and comp



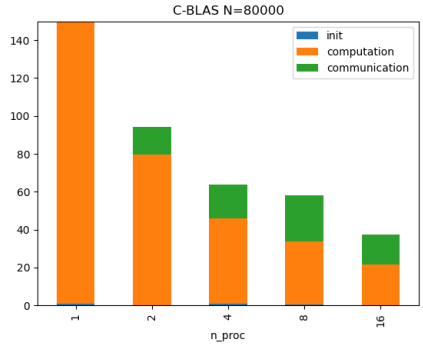

(a) label 1



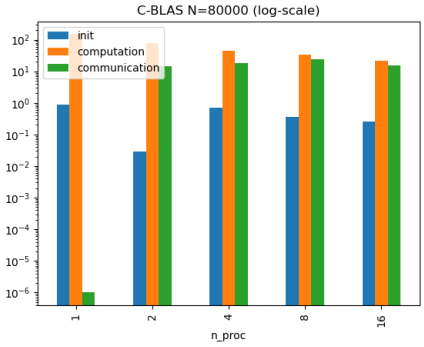

(b) label 2

Figure 8: C-BLAS comm and speedup

## 4.4 N = 80000



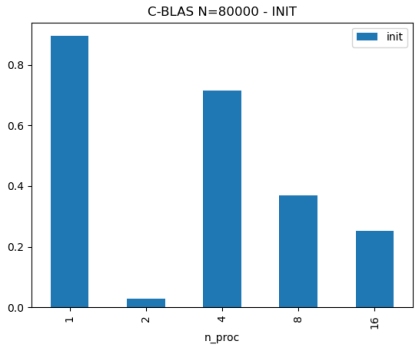

(a) label 1



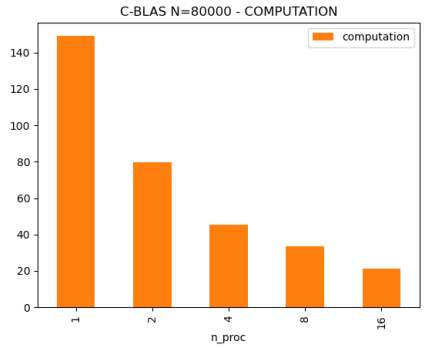

(b) label 2

Figure 9: C-BLAS Scaling

- **Initialization** at 2 cpu nodes on 80000 size results in unexpected low timings. this is

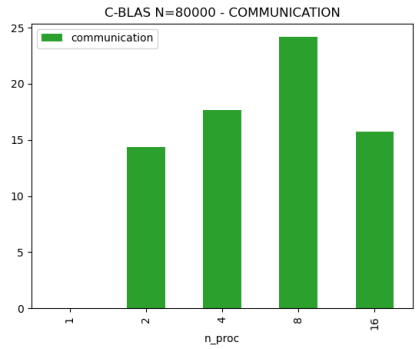



(a) label 1

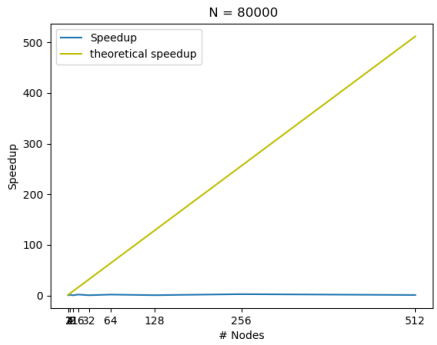



(b) label 2

Figure 10: C-BLAS init and comp




(a) label 1



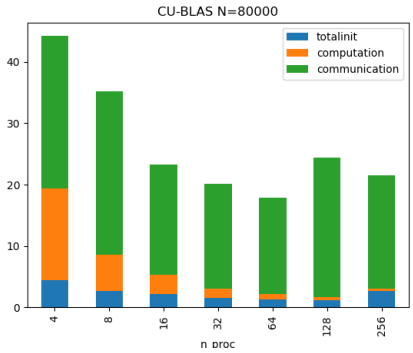

(b) label 2

Figure 11: C-BLAS comm and speedup

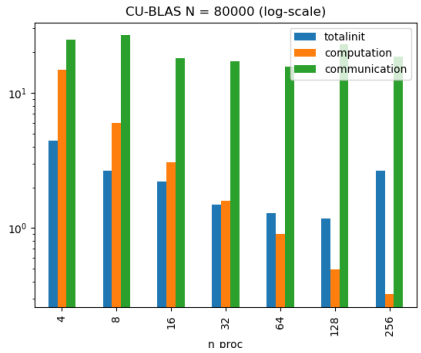
not a coincidence, the initialization has been performed multiple times and the same phenomena has been observed. The size of the matrix (equivalnent to a 55568 size on a single node) probably resonates with the memory architecture of the nodes even though analysis of cache misses (using perf) shows no significant difference between the two cases (40000 vs 80000 size).

## 4.5 CU-BLAS

Scaling is performed mapping processes to GPUs (4 x NVIDIA Ampere A100 GPUs, 64GB per node) assigning 8 cores per process (32 cores Ice Lake at 2.60 GHz per node)

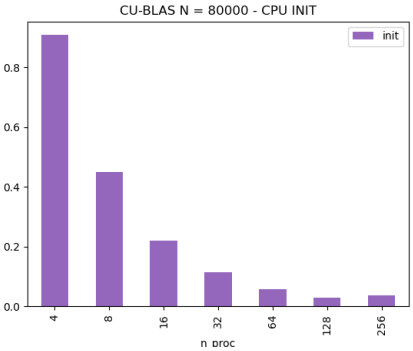



(a) label 1

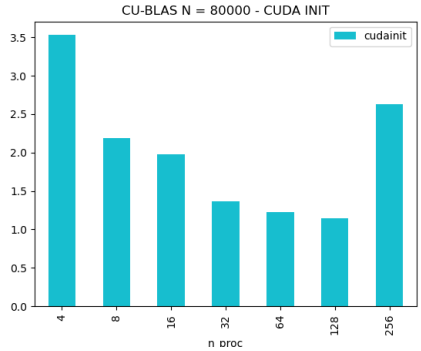



(b) label 2

Figure 12: CU-BLAS Scaling

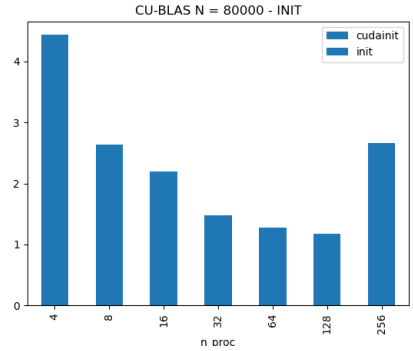



(a) label 1

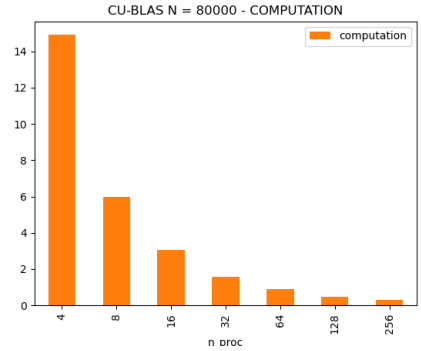



(b) label 2

Figure 13: CU-BLAS init and computation

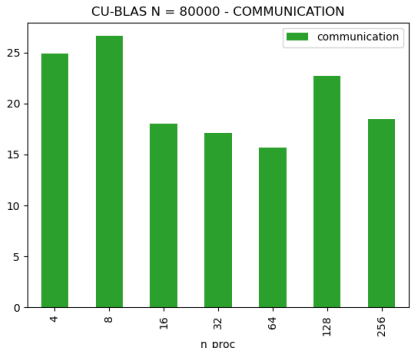



(a) label 1

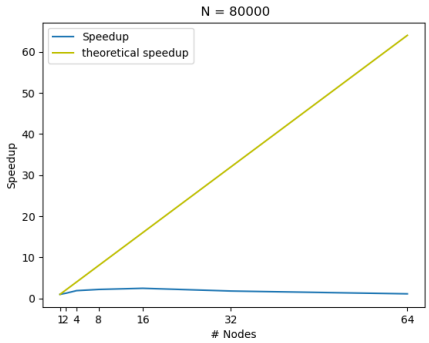



(b) label 2

Figure 14: CU-BLAS init and computation




(a) label 1




(b) label 2

Figure 15: CU-BLAS comm and speedup

# Part II

# Jacobi

## 1 Introduction

### 1.1 Domain Decomposition

Domain is partitioned as in Figure 16. Halo zones are needed since each cell needs to be updated using information from up, down, left and right neighbour.

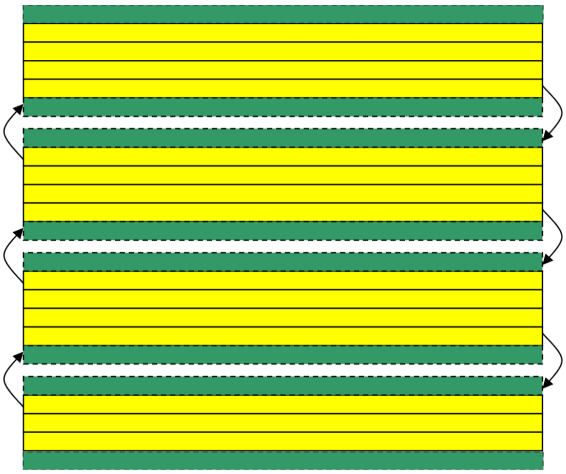


Figure 16: Domain Decomposition

### 1.2 Initialization

Initializationi of the Matrix is performed using OMP / OACC when GPUs are available [17].

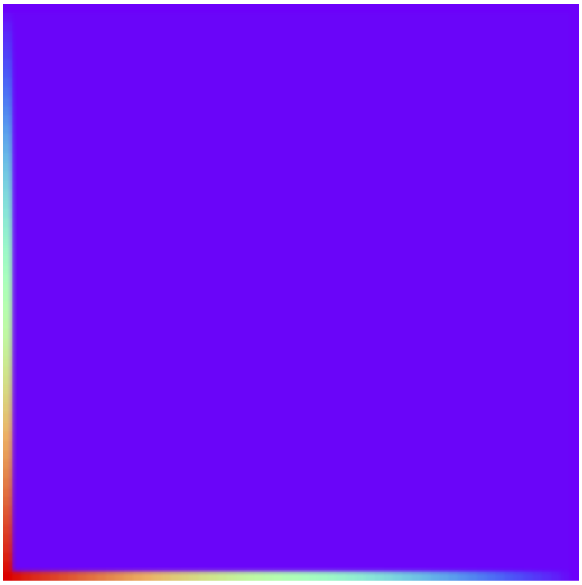


Figure 17: Initial Configuration - Boundary Conditions - 60x60 matrix

## 1.3 Evolution

Evolution of the matrix is performed again using OMP / OACC when GPUs are available.

```
void evolve( double* restrict matrix, double* restrict matrix_new, const int
    N_loc, const size_t dimension )
{
  #pragma acc parallel loop collapse(2) present( matrix, matrix_new )
  for(size_t i = 1 ; i < N_loc-1; ++i ){
    for(size_t j = 1; j < dimension+1; ++j ){
      matrix_new[ ( i * ( dimension + 2 ) ) + j ] = ( 0.25 ) *
         (matrix[ ( ( i - 1 ) * ( dimension + 2 ) ) + j ] +
          matrix[ ( i * ( dimension + 2 ) ) + ( j + 1 ) ] +
          matrix[ ( ( i + 1 ) * ( dimension + 2 ) ) + j ] +
          matrix[ ( i * ( dimension + 2 ) ) + ( j - 1 ) ]);
    }
  }
}
```

## 1.4 MPI Communications

Communication is performed using MPI, two different strategies are implemented: Two-Side and One-Side communication.

### 1.4.1 Two-Side

Communication among processes allows to update the "shared" rows after the update of the "private"[1] ones. Each process is set to send first and last internal (meaning row 1 and

[1]shared and private do not refer to any memory policy but to the rows of the matrix that show up in multiple or just one process

*my_row*) rows to previous and next process defined by the formulas

$$\text{next} = \text{rank} == 0 ? \text{MPI\_PROC\_NULL} : \text{rank} - 1$$

$$\text{prev} = \text{rank} == \text{size} - 1 ? \text{MPI\_PROC\_NULL} : \text{rank} + 1$$

When working with GPUs communication is **Cuda-Aware**, avaoiding the extra copy to Host.

```
void update_boundaries(int rank, int size, double* local_matrix, int my_row,
    int N)
{
    MPI_Request send_request;
    MPI_Status recv_status;

    int tag1=0,tag2=1;

    int prev = rank == 0? MPI_PROC_NULL : rank-1;
    int next = rank == size-1? MPI_PROC_NULL : rank+1;

    // sending last row to rank +1
    #pragma acc host_data use_device( local_matrix )
    {
        MPI_Isend(&local_matrix[(my_row-2)*N],N, MPI_DOUBLE, next, tag1,
    MPI_COMM_WORLD, &send_request);
        // sending first row to rank -1
        MPI_Isend(&local_matrix[N],N, MPI_DOUBLE, prev, tag2, MPI_COMM_WORLD,
     &send_request);

        // reveiving first row from rank -1
        MPI_Irecv(local_matrix,N ,MPI_DOUBLE,prev,tag1,MPI_COMM_WORLD,&
    recv_req_0);
        // receiving last row from rank +1
        MPI_Irecv(&local_matrix[(N_loc-1)*N],N ,MPI_DOUBLE,next,tag2,
    MPI_COMM_WORLD,&recv_req_1);

        MPI_Request array_of_requests[2] = {recv_req_0, recv_req_1};
        MPI_Waitall(2,array_of_requests, MPI_STATUS_IGNORE);
    }
}
```

The communication involves two **Non-Blocking Send** and two **Non-Blocking Receive** for process, This choice avoids any possible deadlock and the **WaitAll** clause assures the process doesn't start a new iteration before getting updated shared rows.

### 1.4.2 One-Side

In the One-Side paradigm, the communication is performed using **MPI Windows**. Four windows are created, two for the upper and lower boundaries of the matrix and two for the new matrix. Such windows are than used in an alternate way to perform **MPI Get** operations.

For window creation infos are used to optimize communication setting the **same size** and **async progress** options.

```
    MPI_Win win_u_n, win_d_n;
```

```
    MPI_Win win_u, win_d;

    MPI_Info info;
    MPI_Info_create(&info);
    MPI_Info_set(info, "same_size", "true");

    MPI_Win win_u_n, win_d_n;
    MPI_Win_create(&matrix_new[dimension+2], (MPI_Aint)(dimension+2) * sizeof
    (double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win_u_n);
    MPI_Win_create(&matrix_new[(N_loc-2)*(dimension+2)], (MPI_Aint)(dimension
    +2) * sizeof(double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &
    win_d_n);

    MPI_Win win_u, win_d;
    MPI_Win_create(&matrix[dimension+2], (MPI_Aint)(dimension+2) * sizeof(
    double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win_u);
    MPI_Win_create(&matrix[(N_loc-2)*(dimension+2)], (MPI_Aint)(dimension+2)
    * sizeof(double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win_d);
```

Both active and passive target communication are tested.

## 1.5 MPI I/O

Parallel write to binary file is also provided. As before, if working on GPUs Cuda-Aware write is perfomed.

```
void mpi_io(int rank, int size, double* local_matrix, int N_loc, int LDA,
    char* fname_snap, MPI_File file)
{
    char filename[256];
    sprintf(filename, "%s.bin", fname_snap);

    MPI_Offset offset = ((N_loc-2) * rank + (LDA-2)%size*(rank>=(LDA-2)%size)
    ) * LDA * sizeof(double);

    MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE | MPI_MODE_WRONLY
    , MPI_INFO_NULL, &file);

    // Set the file pointer to the correct position
    MPI_File_seek(file, offset, MPI_SEEK_SET);

    // Set the file view
    MPI_File_set_view(file, offset, MPI_DOUBLE, MPI_DOUBLE, "native",
    MPI_INFO_NULL);

    // Write the local matrix block to the file
    #pragma acc host_data use_device( local_matrix )
    MPI_File_write(file, &local_matrix[LDA], (N_loc-2)*LDA, MPI_DOUBLE,
    MPI_STATUS_IGNORE);

    // Close the file
    MPI_File_close(&file);
}
```

The final image [18] can be showed using *Gnuplot*.

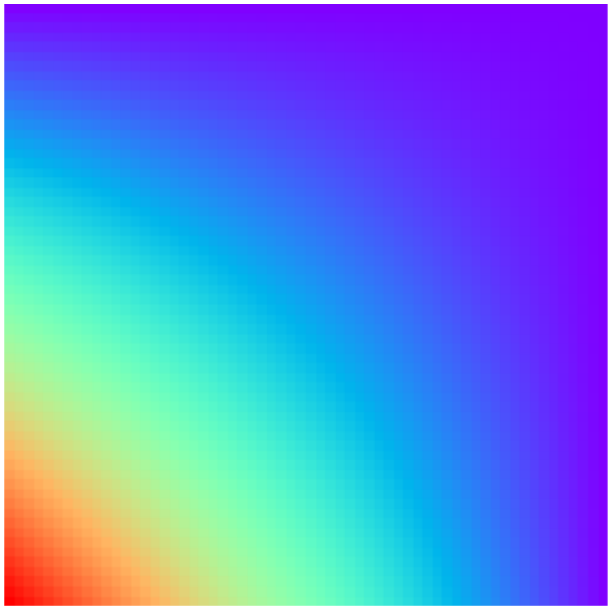


Figure 18: Final Configuration - 2000 iterations - 60x60 matrix

### 1.6 Computation

The final algorithm looks like this:

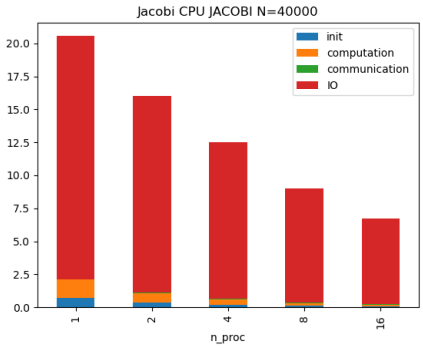
**Algorithm 2** Jacobi

```
for i = 1, 2, ..., iterations do
    Evolve(M,M_new,N_loc,LDA)
    Swap(M,M_new)
    update_halo(M,rank,size)
end for
```

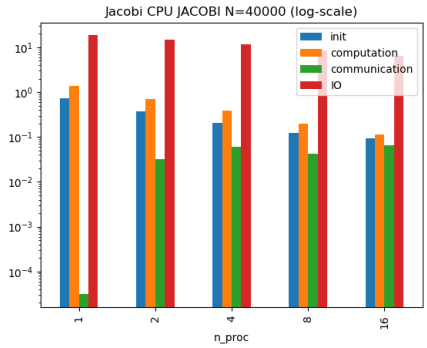
## 2 Scaling Results

### 2.1 CPU

Scaling is performed mapping processes to nodes (2x56 cores Intel Sapphire Rapids at 2.00 GHz) using all threads available (112 - no hyperthreading).

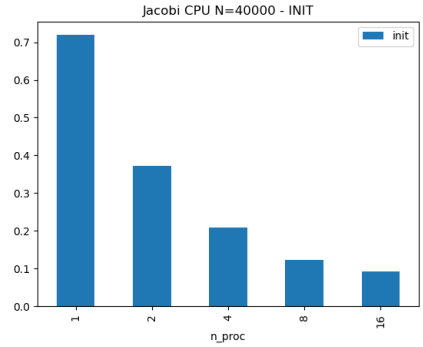



(a) label 1

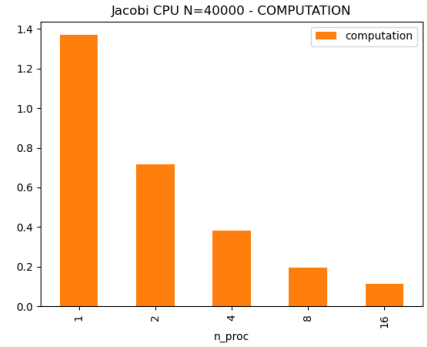



(b) label 2

Figure 19: CPU Scaling

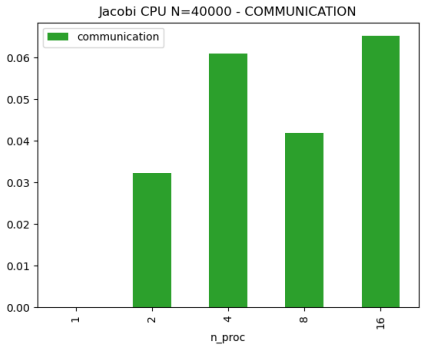



(a) label 1

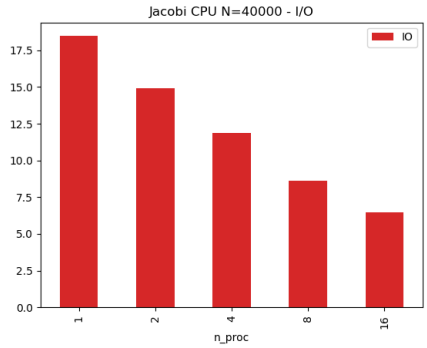



(b) label 2

Figure 20: CPU Scaling

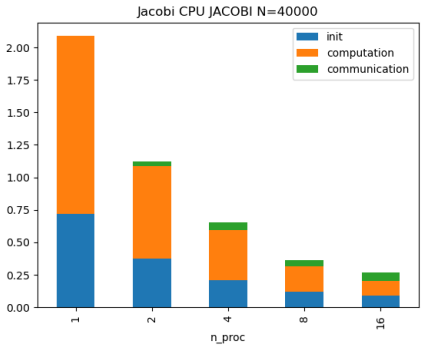



(a) label 1

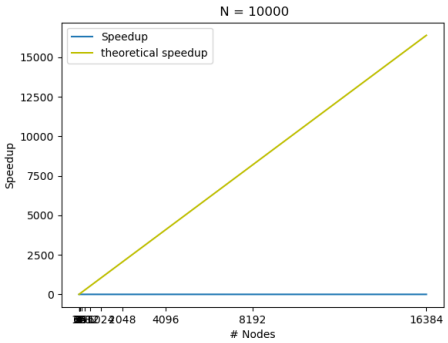



(b) label 2

Figure 21: CPU Scaling




(a) label 1

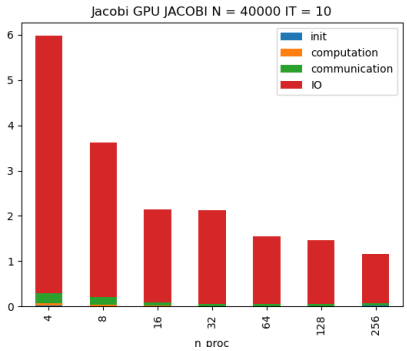



(b) label 2

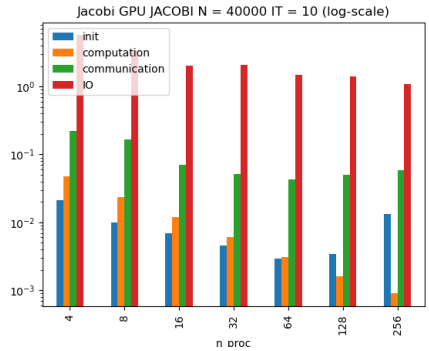
Figure 22: CPU Scaling

## 2.2 GPU

Scaling is performed mapping 4 processes per node such that each one is responsible for one GPUs (4 x NVIDIA Ampere A100 GPUs, 64GB per node) assigning 8 cores per process (32 cores Ice Lake at 2.60 GHz per node)

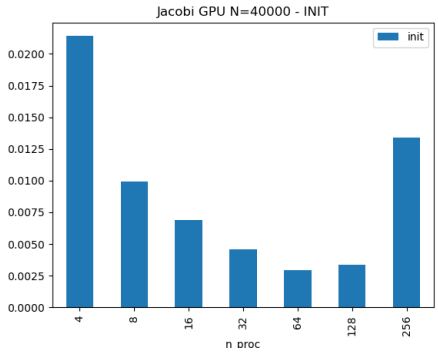



(a) label 1

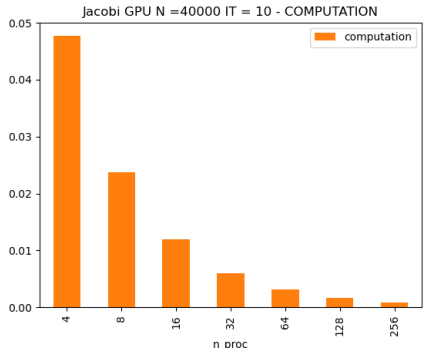



(b) label 2

Figure 23: GPU Scaling

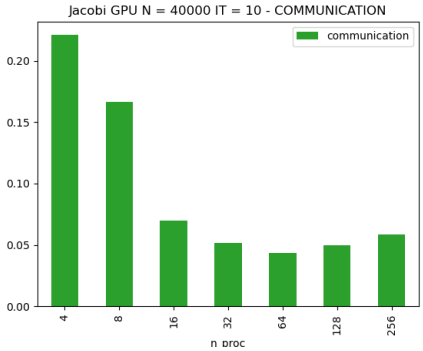



(a) label 1

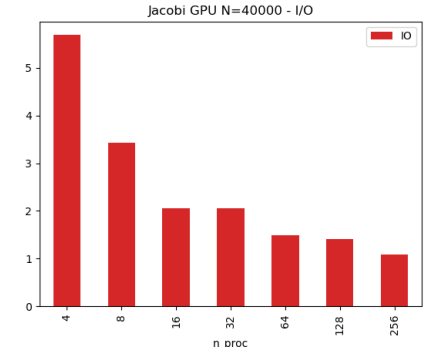



(b) label 2

Figure 24: GPU Scaling

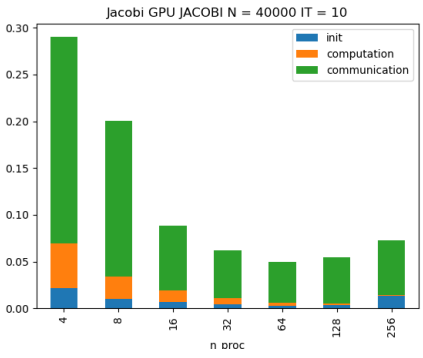



(a) label 1




(b) label 2

Figure 25: GPU Scaling




(a) label 1

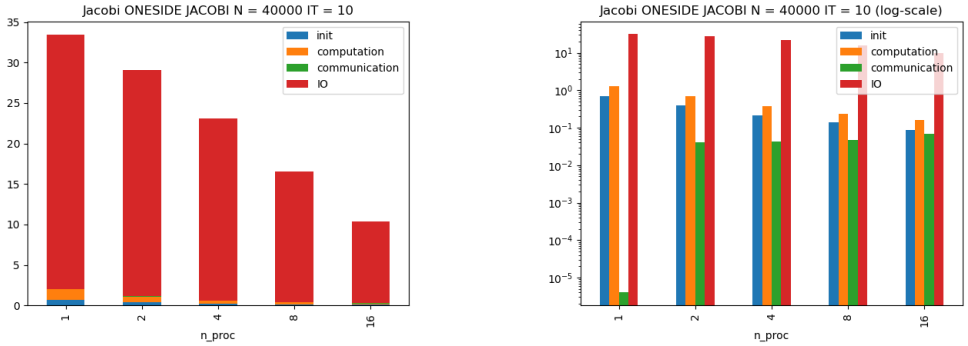
Figure 26: CPU Scaling

## 2.3 OneSide Communcation

### 2.3.1 Fence

Communication is then performed using active target communication, through the use of fences.

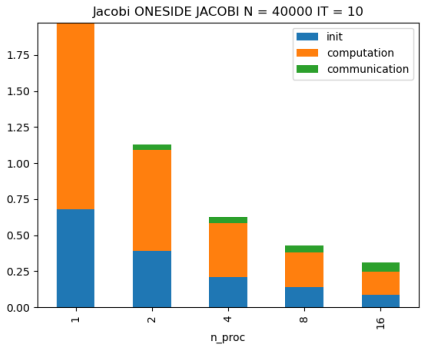
```
void update_boun_oneside_fence(int rank,int size,double* matrix_new,int N_loc
    ,int dimension,MPI_Win win_u,MPI_Win win_d)
{
    int prev = rank == 0? MPI_PROC_NULL : rank-1;
    int next = rank == size-1? MPI_PROC_NULL : rank+1;

    MPI_Win_fence(MPI_MODE_NOPUT, win_d);
    MPI_Win_fence(MPI_MODE_NOPUT, win_u);
    MPI_Get(matrix_new, dimension+2, MPI_DOUBLE, prev, 0, dimension+2,
    MPI_DOUBLE, win_d);
    MPI_Get(&matrix_new[(N_loc-1)*(dimension+2)], dimension+2, MPI_DOUBLE,
    next, 0, dimension+2, MPI_DOUBLE, win_u);
    MPI_Win_fence(MPI_MODE_NOPUT, win_u);
    MPI_Win_fence(MPI_MODE_NOPUT, win_d);

}
```



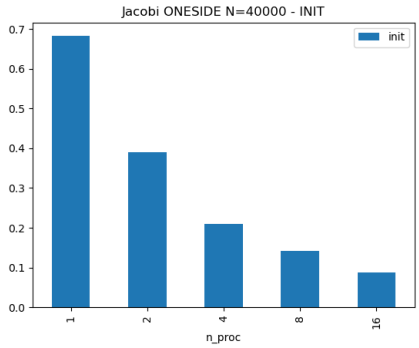

(a) label 1

(b) label 2

Figure 27: ONESIDE Fence Scaling

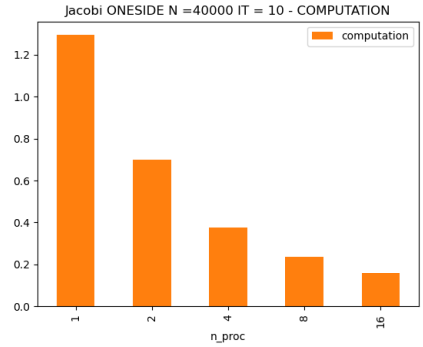



(a) label 1

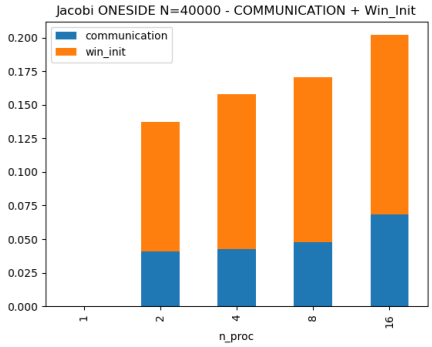



(b) label 2

Figure 28: ONESIDE Fence Scaling

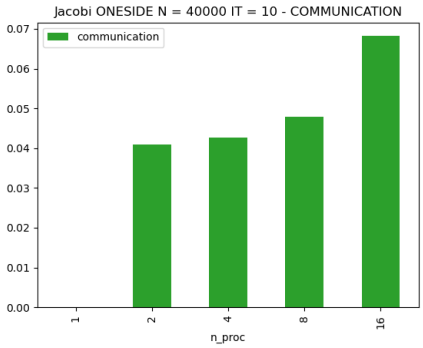



(a) label 2

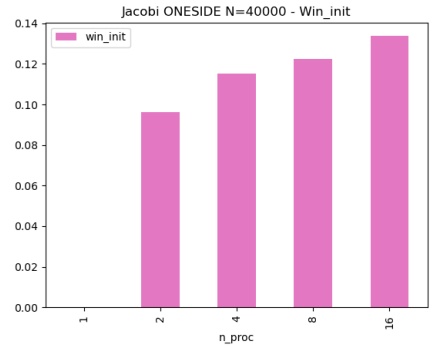



(b) label 1

Figure 29: ONESIDE Fence Scaling




(a) label 2



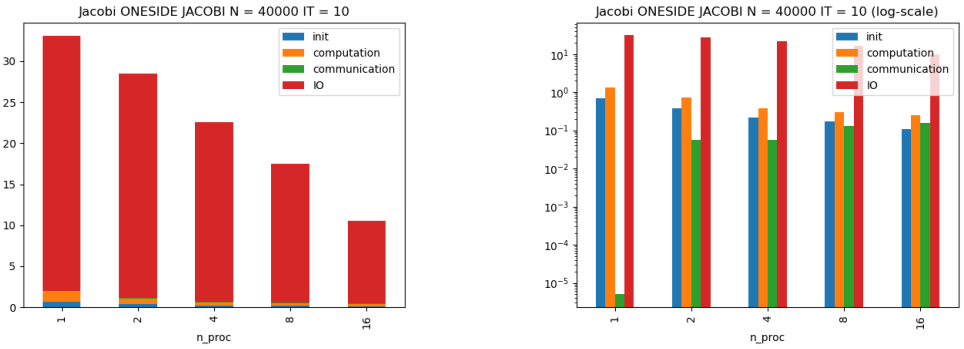

(b) label 1

Figure 30: ONESIDE Fence Scaling

### 2.3.2 Lock

Communication is then performed using passive target communication, the process locks the window, gets the data and unlocks it. Beacuse of the passive target communication, one has to guarantee that the data is available before the process tries to get it, this is ensured by the use of a MPI_Barrier.

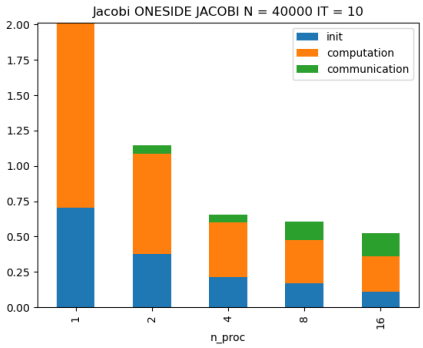
```
void update_boun_oneside_lock(int rank,int size,double* matrix_new,int N_loc,
    int dimension,MPI_Win win_u,MPI_Win win_d)
{
    MPI_Barrier(MPI_COMM_WORLD);

    int prev = rank == 0? MPI_PROC_NULL : rank-1;
    int next = rank == size-1? MPI_PROC_NULL : rank+1;

    if (rank!=0){
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE,prev,MPI_MODE_NOCHECK, win_d);
        MPI_Get(matrix_new, dimension+2, MPI_DOUBLE, prev, 0, dimension+2,
    MPI_DOUBLE, win_d);
        MPI_Win_unlock(prev, win_d);
    }
    if(rank!=size-1){
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE,next,MPI_MODE_NOCHECK, win_u);
        MPI_Get(&matrix_new[(N_loc-1)*(dimension+2)], dimension+2, MPI_DOUBLE
    , next, 0, dimension+2, MPI_DOUBLE, win_u);
        MPI_Win_unlock(next, win_u);
    }
}
```



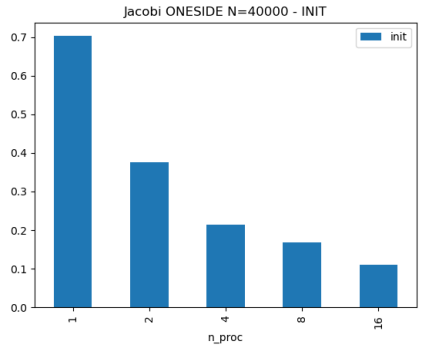

(a) label 1

(b) label 2

Figure 31: ONESIDE Lock Scaling

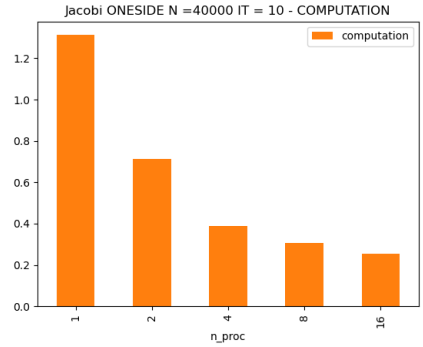



(a) label 1

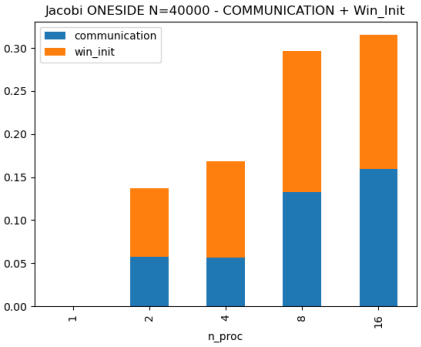



(b) label 2

Figure 32: ONESIDE Lock Scaling

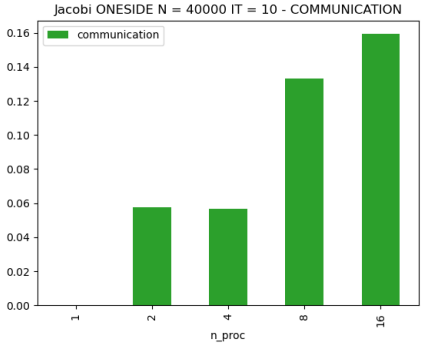



(a) label 2

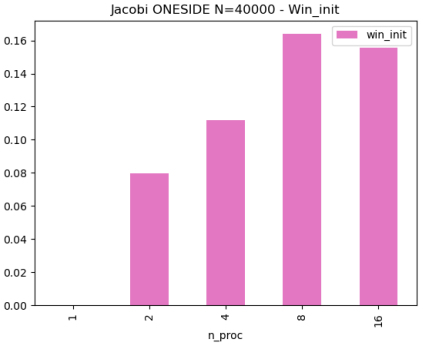



(b) label 1

Figure 33: ONESIDE Lock Scaling




(a) label 2

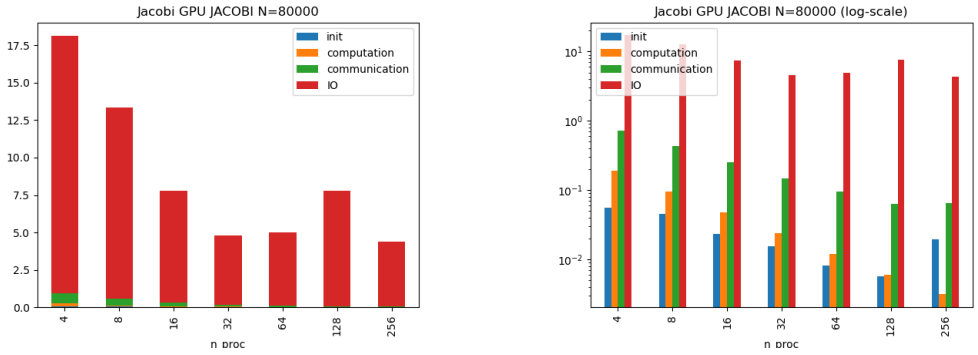



(b) label 1

Figure 34: ONESIDE Lock Scaling

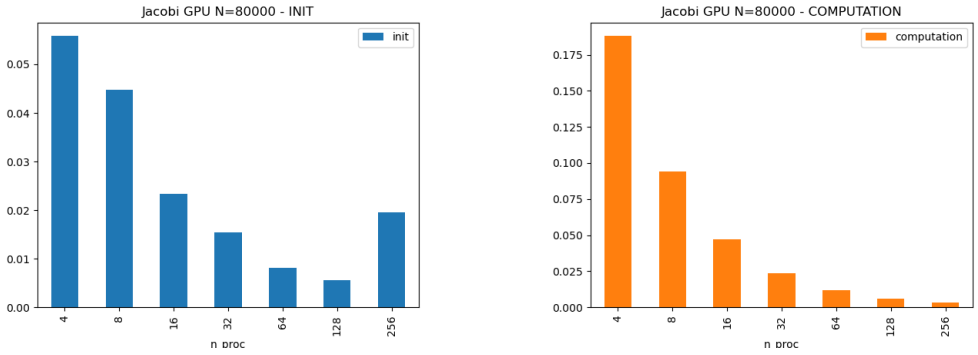
# Appendix A More Plots

# Appendix B GPU JACOBI 80000



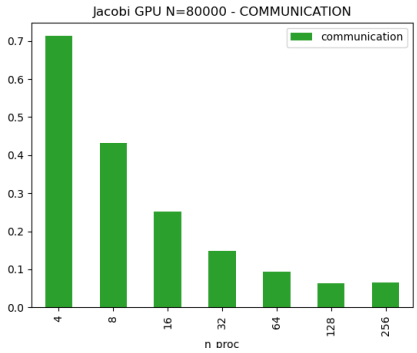

(a) label 1 (b) label 2

Figure 35: GPU Scaling

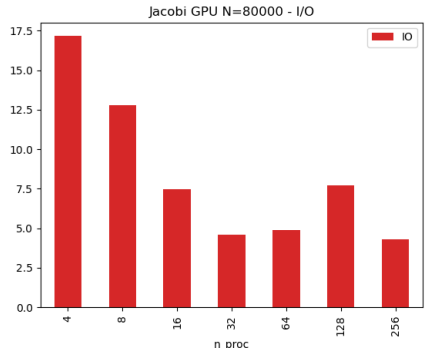



(a) label 1 (b) label 2

Figure 36: GPU Scaling

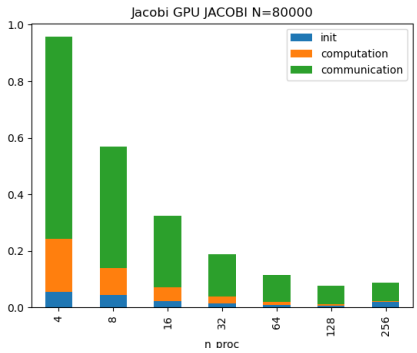



(a) label 1

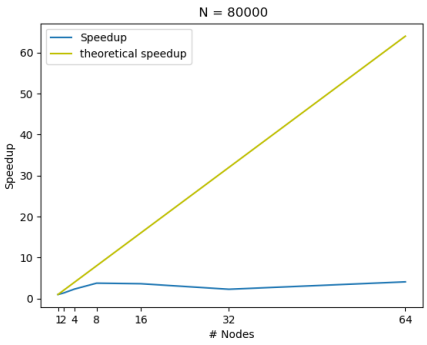



(b) label 2

Figure 37: GPU Scaling




(a) label 1




(b) label 2

Figure 38: CPU Scaling