Advanced HPC - Final assignment

Tommaso Tarchi

October 8, 2024

University of Trieste

Matrix-matrix multiplication

List of implementations

1. Naive implementation ("by hand").
2. Using `BLAS dgemm` function.
3. Using `cuBLAS dgemm` function.

General algorithm

1. A, B and C are allocated as blocks of n_{rows} rows across processes (called A_{loc}, B_{loc} and C_{loc}).
2. Each process initializes A_{loc} and B_{loc} randomly;
3. (If using cuBLAS) A_{loc} is moved to device.
4. Considering each C_{loc} as divided into n_{procs} adjacent blocks of $n_{rows} \times n_{cols}$ elements, the following points are iterated n_{procs} times:
   - 4a. each process communicates to all other processes only B_{loc}'s data needed to compute the i-th block of C_{loc};
   - 4b. each process gathers data coming from all other processes into a single matrix, called B_{col};
   - 4c. (if using cuBLAS) B_{col} is moved to device;
   - 4d. each process computes the i-th block of C_{loc}.
5. (If using cuBLAS) C_{loc} computed on device is moved back to host.

Implementation details

- For random initialization of A and B, a unique seed was computed randomly by process 0 and broadcasted to all other processes, which then set different seeds for each one of their thread.
- MPI communications were performed using `MPI_Allgatherv()`.
- When using BLAS and cuBLAS, we accumulated the computed entries of C_{loc} directly on the memory allocated for C_{loc}, by combining the pointer to matrix and stride arguments.
- When using cuBLAS, we avoided transpositions by passing as arguments B and A in this order, computing in this way $B_{col}^T \times A_{loc}^T$.

Profiling

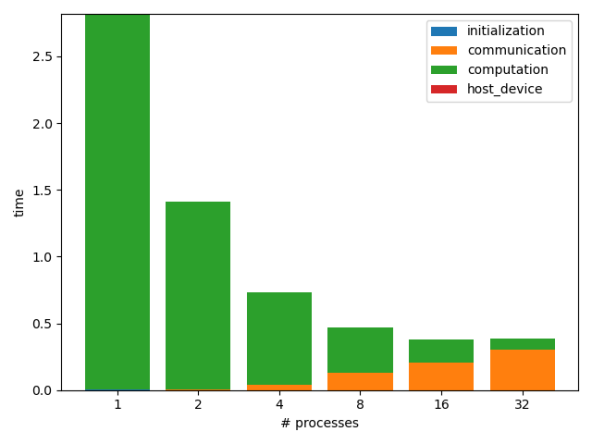
Times measured:

1. **Matrices initialization**, including:
   - random initialization of matrices A_{loc} and B_{loc}.
2. **Communication**, including:
   - creation of all B_{block}'s;
   - all executions of `MPI_Allgatherv()`.
3. **Computation**, including:
   - all executions of "by hand" product or `dgemm()`.
4. **Host-device data movements** (only for cuBLAS code), including:
   - initial copy of A_{loc} from host to device;
   - copy of all B_{col}'s from host to device;
   - final copy of C_{loc} from device to host.

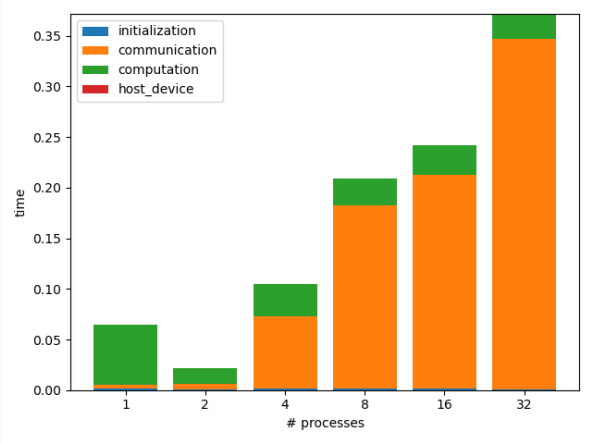
Profiling

- All measured times were averaged over all processes.
- We placed one process per socket in non GPU-accelerated programs (i.e. two processes per node) and one per device in GPU-accelerated program (i.e. four processes per node).

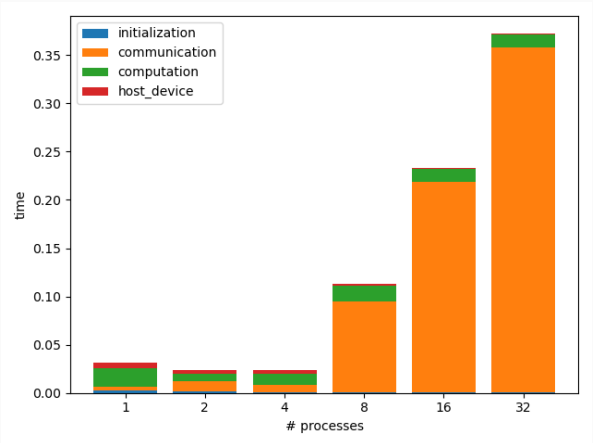
Results - size 1000×1000



(a) "naive" - openMP threads: 20

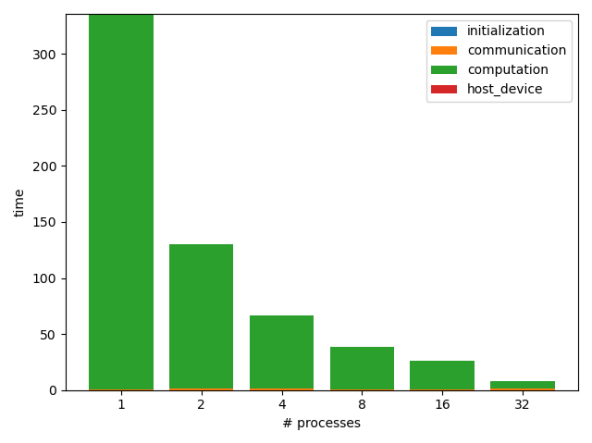


(b) BLAS; 20 openMP threads

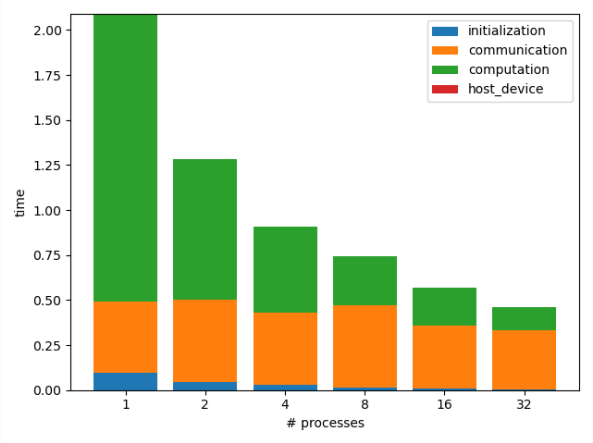


(c) cuBLAS; 8 openMP threads

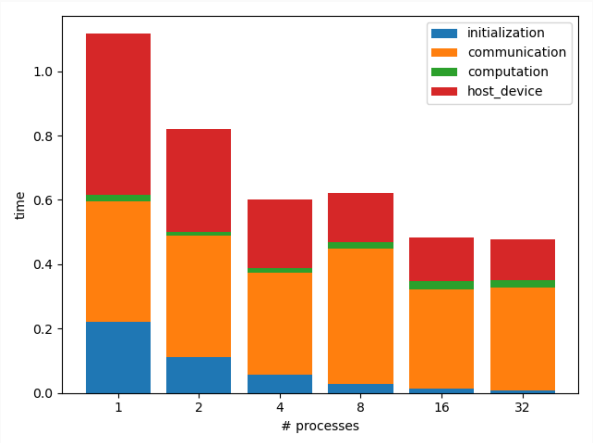
Results - size 10000×10000



(a) "naive" - openMP threads: 20

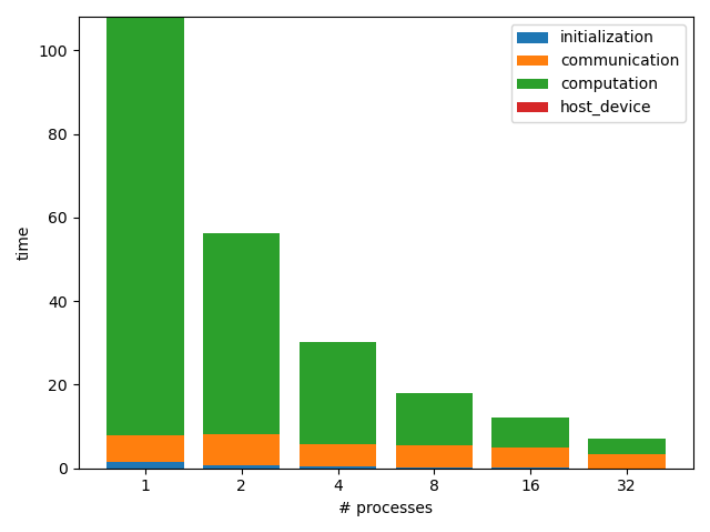


(b) BLAS; 20 openMP threads

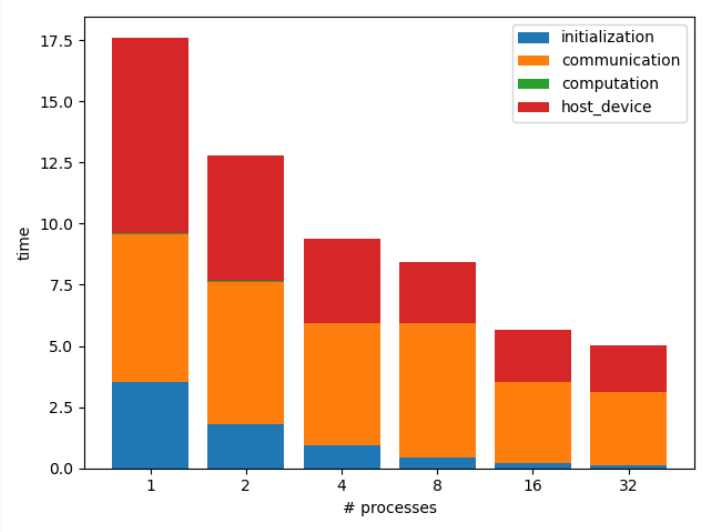


(c) cuBLAS; 8 openMP threads

Results - size 40000×40000



(a) BLAS; 20 openMP threads



(b) cuBLAS; 8 openMP threads

Observations

- Even if matrix block creation should scale, the total time for communications does not. Likely, this is because of:
  - increasing complexity in orchestrating collective communications;
  - increasing physical distance between processes.

  This is particularly clear from the increasing bad communication performance as the matrix size decreases (smaller blocks but same overhead).
- Host-device data movement time does not scale perfectly, probably because of some overhead.
- When using cuBLAS the limits to performance become MPI and host-device communications.

Jacobi method for Laplace equation

List of implementations

1. OpenACC with CUDA-unaware MPI.
2. OpenACC with CUDA-aware MPI.
3. MPI with blocking communications (`MPI_Sendrecv`).
4. MPI with non-blocking communications (`MPI_Isend` and `MPI_Irecv`).
5. MPI with remote memory access (`MPI_Put`).

All programs were further parallelized using openMP.

OpenACC - algorithm

1. Each process allocates its own portion of the grid. Also, an identical auxiliary matrix to store updated cells state is allocated.
2. Each process initializes its own cells.
3. Both main and auxiliary grids are moved to device.
4. For the required number of iterations, each process repeats the following:
   - 4a. communicates to the bordering processes the corresponding neighbor rows from host to host (CUDA-unaware) or from device to device (CUDA-aware);
   - 4b. (only for CUDA-unaware) received boundaries are updated on device;
   - 4c. evolves its own cells on device on the auxiliary grid, then swaps pointers on both host and device to update main grid;
   - 4d. (only for CUDA-unaware) updated border cells are copied back to host.
5. The final grid is copied back to host.

OpenACC - implementation details

- Communication of boundaries between processes was performed using `MPI_Sendrecv()`.
- Data movements were managed by placing an openACC data region around the for loop performing communications and evolution of the grid, and using clause `copy` for main and auxiliary grid.
- In the CUDA-unaware version we used the `update` clause to copy borders from host to device and vice versa.
- For CUDA-aware comunications we used an openACC region with clauses `host_data` and `use_device` to "wrap" MPI calls.

OpenACC - profiling

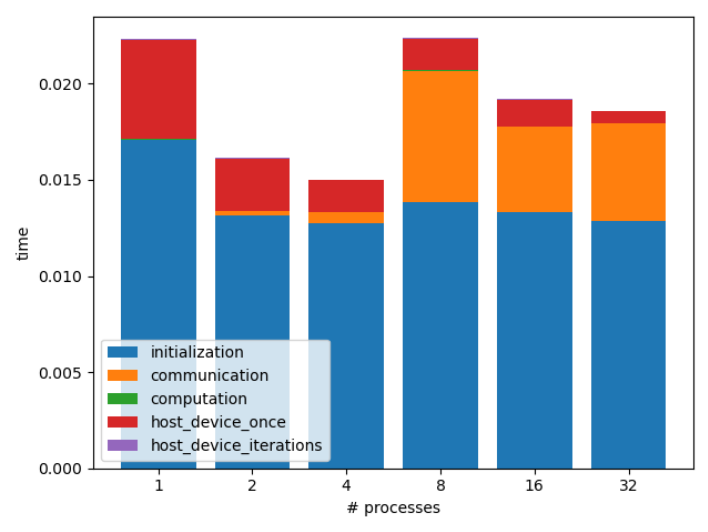
Times measured:

1. **System initialization**, including:
   - allocation and initialization of main and auxiliary matrices.
2. **One-time host-device data movements**, including:
   - initial copy of matrices from host to device;
   - final copy of main matrix from device back to host.
3. **Communication**, including:
   - all executions of `MPI_Sendrecv()`.
4. **Computation**, including:
   - all system updates on auxiliary matrix;
   - all pointer swaps for main matrix update.
5. **In-loop host-device data movements**, including:
   - all incoming borders updates after communications (host to device);
   - all outgoing borders updates after computation (device to host).

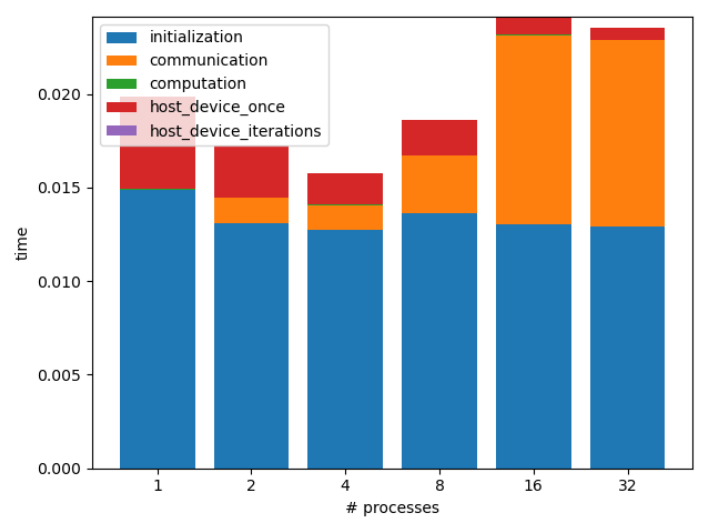
OpenACC - profiling

- Initialization and one-time host-device transfer times were averaged over all processes; while other times were averaged over iterations first and processes then.
- We placed one process per device (i.e. four processes per node).

OpenACC - results - size 1200×1200

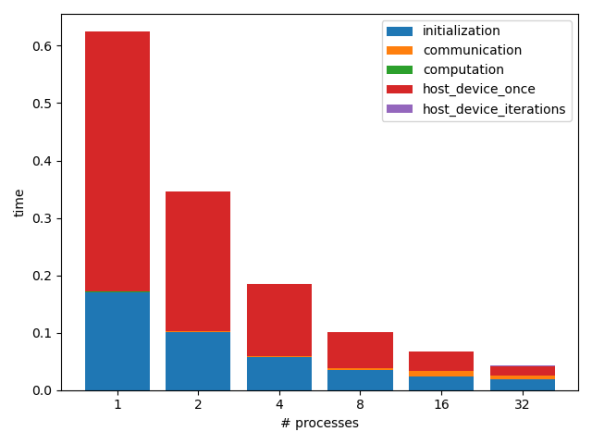


(a) CUDA-unaware MPI; 8 openMP threads

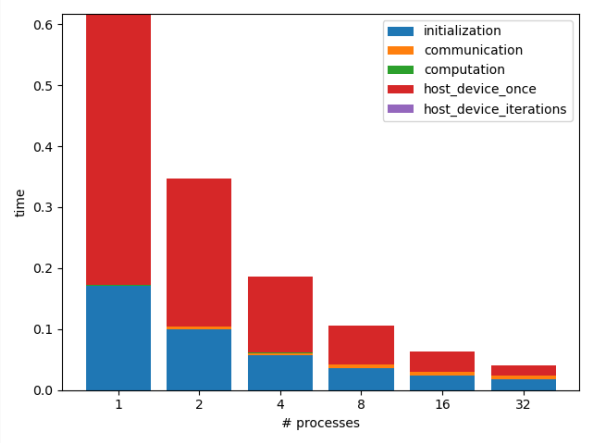


(b) CUDA-aware MPI; 8 openMP threads

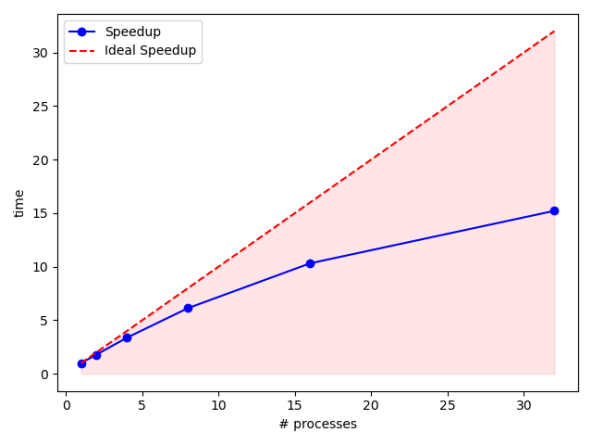
OpenACC - results - size 12000×12000



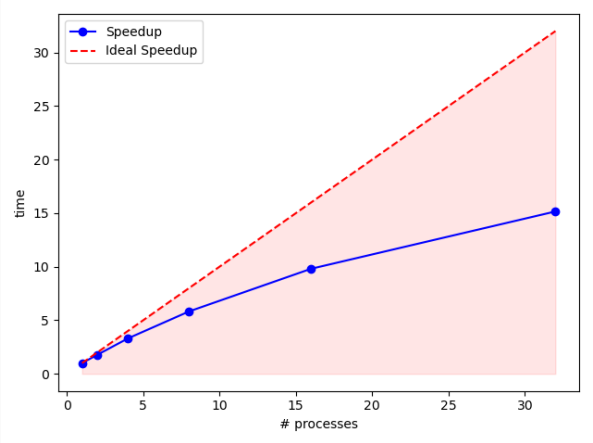
(a) CUDA-unaware MPI; 8 openMP threads



(b) CUDA-aware MPI; 8 openMP threads

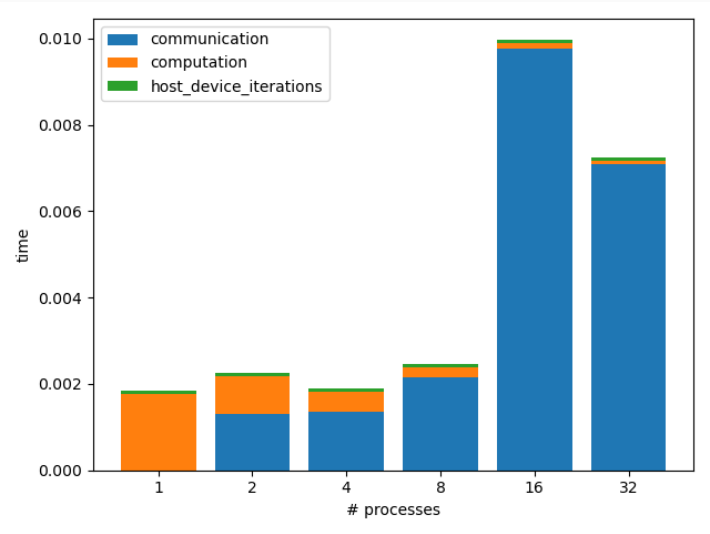


(c) CUDA-unaware MPI; 8 openMP threads

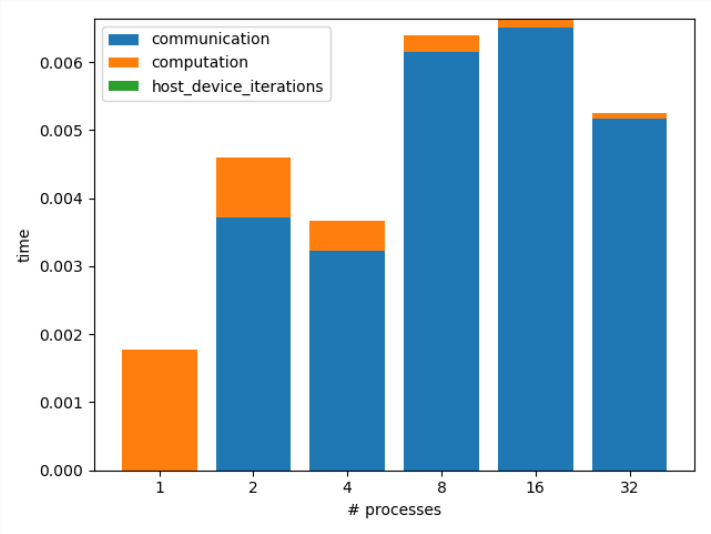


(d) CUDA-aware MPI; 8 openMP threads

OpenACC - results - size 12000×12000 (only iterative part)

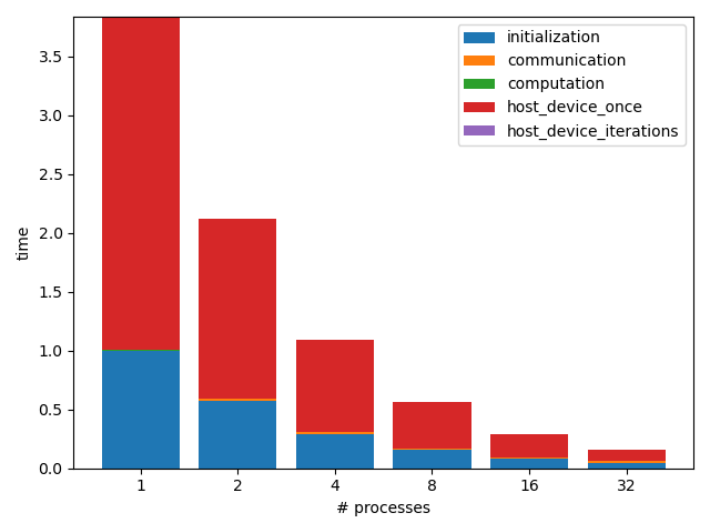


(a) CUDA-unaware MPI; 8 openMP threads

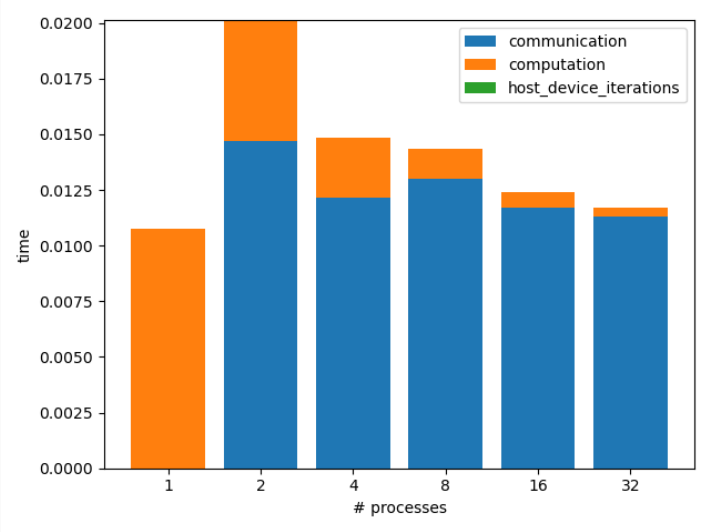


(b) CUDA-aware MPI; 8 openMP threads

OpenACC - results - size 30000×30000



(a) CUDA-aware MPI; 8 openMP threads - all times



(b) CUDA-aware MPI; 8 openMP threads - only ietrative times

OpenACC - observations

- In this particular case the benefit from performing CUDA-aware MPI communications is not really significant.
- The possibility of scaling is killed by the inefficiency of MPI communications. The larger the number of iterations the stronger this effect.
- MPI remote memory access could enhance scalability of the program (see next section).

MPI RMA - algorithm

1. Each process allocates its own portion of the grid and two additional arrays to store the upper and lower boundaries separately. Also, an auxiliary grid to store updated cells state is allocated.
2. Each process initializes its own cells.
3. For the required number of iterations, each process repeats the following:
   - 3a. opens two windows on its border arrays to allow neighbor processes to write directly;
   - 3b. writes directly to the neighbor processes' border arrays the corresponding neighbor rows (i.e. first and last), while evolves internal cells on the auxiliary grid;
   - 3c. closes the opened windows;
   - 3d. evolves cells on the first and last rows on the auxiliary grid and then swaps pointers to update main grid.

MPI RMA - implementation details

- For RMA operations we used `MPI_Put()` for direct writing.
- The boundary arrays were allocated contextually to window creation using `MPI_Win_allocate()`.
- We created for each process two `MPI Groups` containing one the previous and one the following process, and used them to expose the windows exclusively to the interested processes.
- We used `MPI_Win_post()`, `MPI_Win_start()`, `MPI_Win_complete()` and `MPI_Win_wait()` to make synchronization less "rigid"; in particular we evolved the internal cells before calling `MPI_Win_wait()`, so the evolution of these cells could be performed even if the boundaries had not been received yet.

MPI RMA - profiling

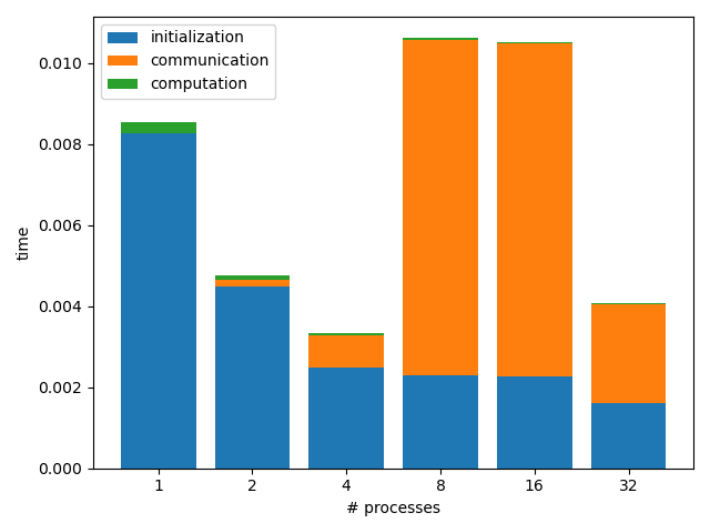
Times measured:

1. **System initialization**, including:
   - allocation of main and auxiliary matrices and window allocation for boundaries;
   - main and auxiliary matrices and extreme boundaries initialization.
2. **Communication**, including:
   - all MPI direct put on neighbor processes's windows execution (i.e. `MPI_Win_start()`, `MPI_Put()` and `MPI_Win_complete()`);
   - all waits for MPI direct put on process's own windows to complete (i.e. `MPI_Win_wait()`).
3. **Computation**, including:
   - all system updates of internal rows on auxiliary matrix;
   - all system updates of external rows on auxiliary matrix;
   - all pointer swaps for main matrix update.

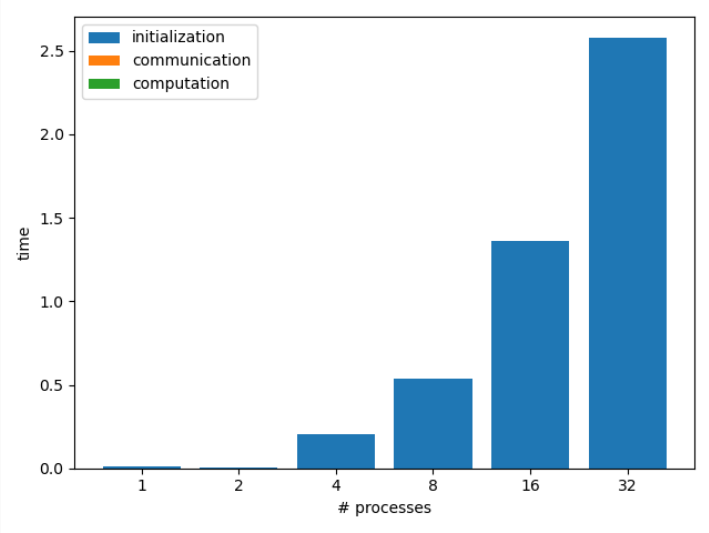
MPI RMA - profiling

- Initialization time was averaged over all processes; while other times were averaged over iterations first and processes then.
- We placed one process per socket (i.e. two processes per node).
- **Measured communication time is not the total time for communication**: it is the additional time needed for communication w.r.t. computing time (**same goes for version with non-blocking MPI**).

MPI RMA - results - size 1200×1200

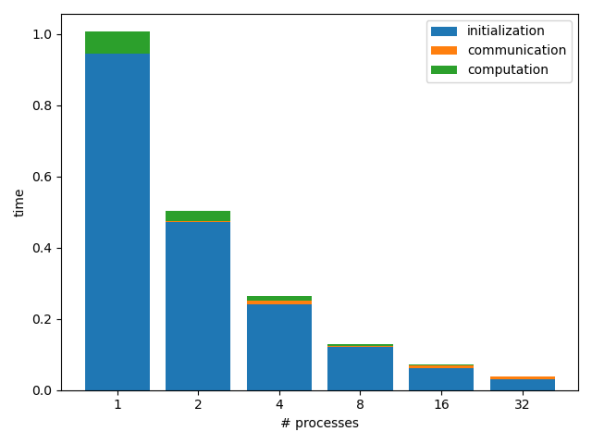


(a) blocking MPI; 20 openMP threads

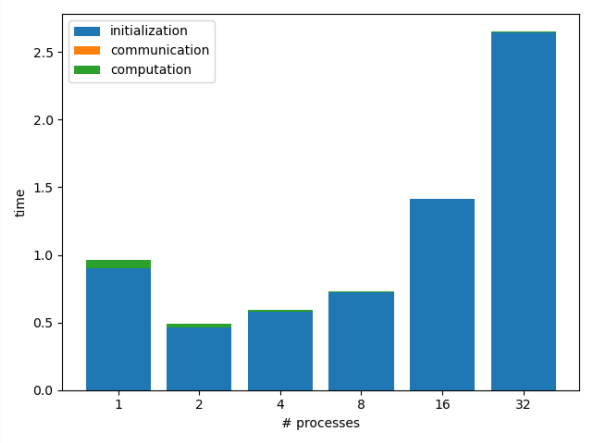


(b) MPI RMA; 20 openMP threads

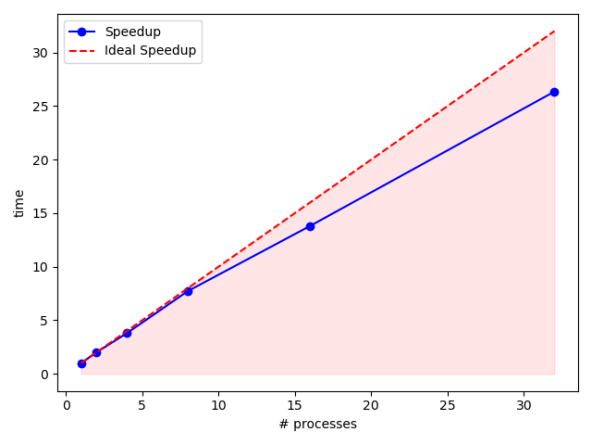
MPI RMA - results - size 12000×12000



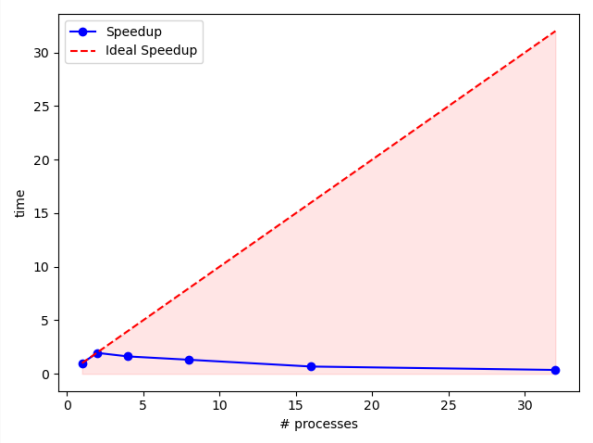
(a) blocking MPI; 20 openMP threads



(b) MPI RMA; 20 openMP threads

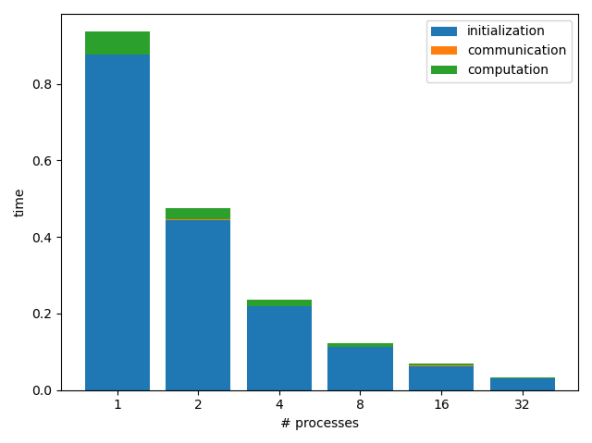


(c) blocking MPI; 20 openMP threads

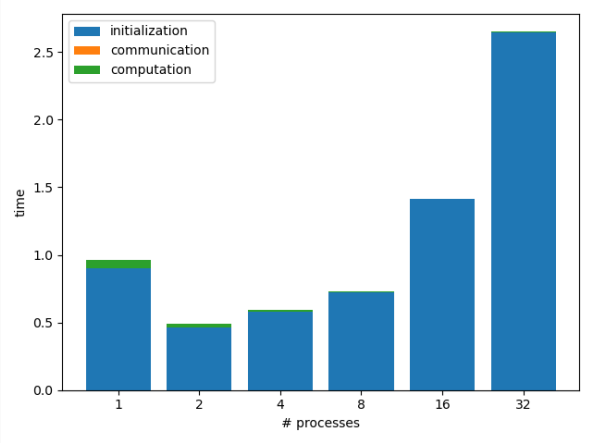


(d) MPI RMA; 20 openMP threads

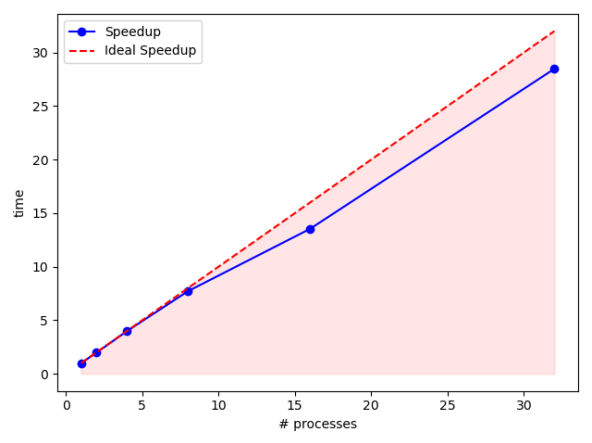
MPI RMA - results - size 12000×12000



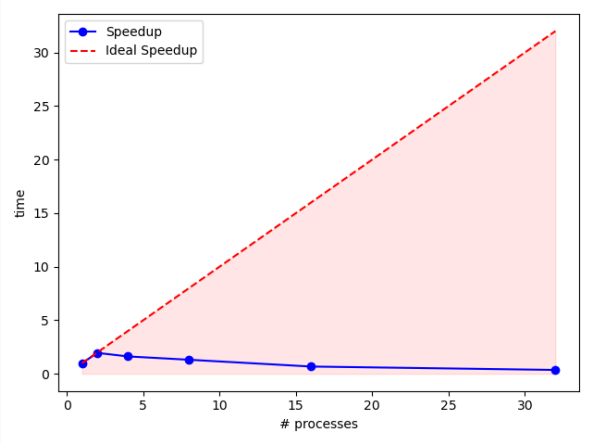
(a) non-blocking MPI; 20 openMP threads



(b) MPI RMA; 20 openMP threads

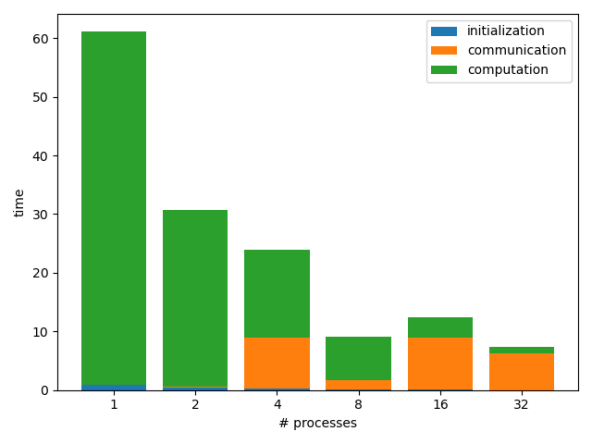


(c) non-blocking MPI; 20 openMP threads

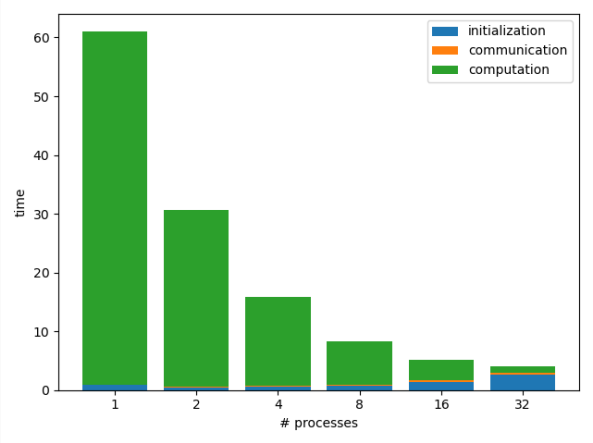


(d) MPI RMA; 20 openMP threads

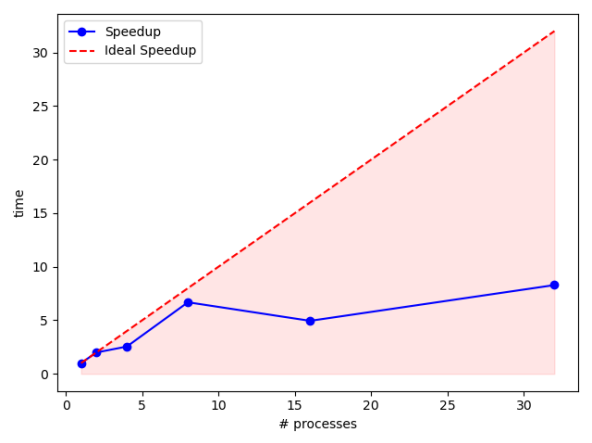
MPI RMA - results - size 12000×12000 (1000 iterations)



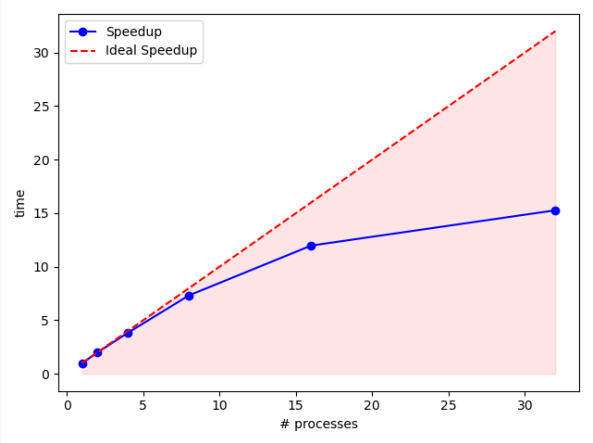
(a) blocking MPI; 20 openMP threads



(b) MPI RMA; 20 openMP threads

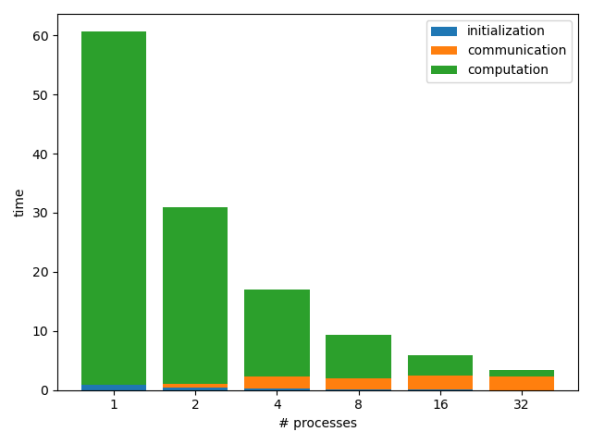


(c) blocking MPI; 20 openMP threads

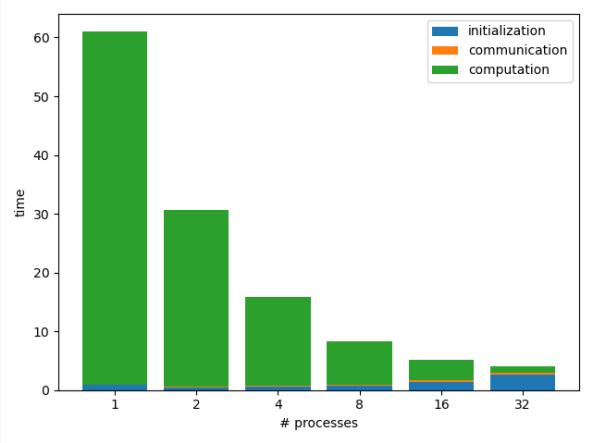


(d) MPI RMA; 20 openMP threads

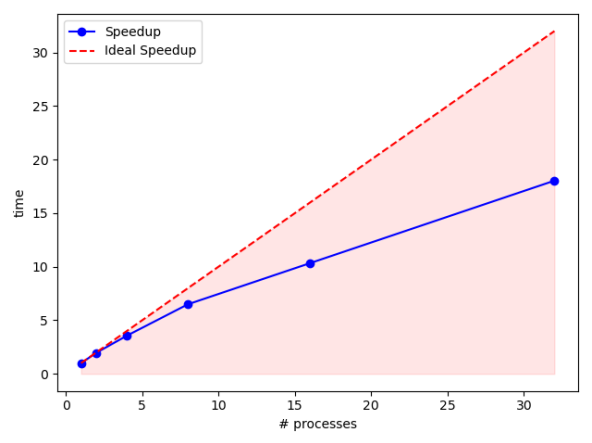
MPI RMA - results - size 12000 × 12000 (1000 iterations)



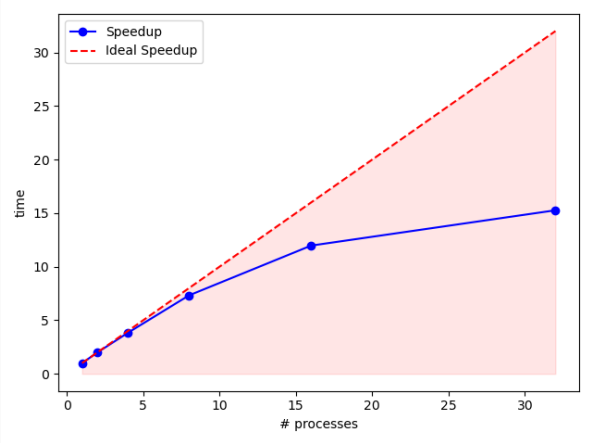
(a) non-blocking MPI; 20 openMP threads



(b) MPI RMA; 20 openMP threads



(c) non-blocking MPI; 20 openMP threads



(d) MPI RMA; 20 openMP threads

MPI RMA - observations

- Shared windows initialization is very costly, but makes communications much more efficient.
- The higher the number of iterations, the more it's beneficial to use MPI RMA (**RMA "moves" the computational cost from communications to initialization**). This is true w.r.t. non-blocking communications as well.
- Despite being much more efficient than blocking ones, non-blocking communications still limit scaling for higher numbers of processes.
- Cost of shared windows initialization grows with the number of processes.

Thank you!