ADVANCED HIGH PERFORMANCE COMPUTING
FINAL ASSIGNMENT

Msc in Data Science and Scientific Computing
University of Trieste

Author
Tommaso Tarchi
Trieste
7-15-2024

Contents

1 CUDA assignment

1.1 Introduction

In this assignment we want to compare performance and scaling of three MPI-parallelized programs for floating point matrix-matrix multiplication:

1. "naive" implementation (i.e. writing the function for multiplication from scratch);
2. using `dgemm` function from BLAS library;
3. using `dgemm` function from cuBLAS library (i.e. the GPU-accelerated "version" of point 2.).

1.2 Parallel algorithm

The algorithm used to distribute matrix-matrix multiplication is similar for all three programs, differences are in the way multiplication is performed on the single MPI process and in data transfer for GPU-accelerated version.

Our goal is to perform:

$$C = AxB,$$

where A, B and C are three square matrices of double precision floating point numbers.

We call n_{procs} the number of MPI processes.

To make communications efficient by performing the smallest number of data transfers possible, the following algorithm was adopted:

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 its 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 $n_{rows}xn_{cols}$ elements, the following points are iterated n_{procs} times:
   - each process communicates to all other processes only B_{loc}'s data needed to compute the i-th block of C_{loc};
   - each process gathers data coming from all other processes into a single matrix, called B_{col};
   - if using cuBLAS, B_{col} is moved to device;
   - each process computes the i-th block of C_{loc} (if using cuBLAS, C_{loc} is computed directly on device).
5. If using cuBLAS, C_{loc} computed on device is moved back to host.

1.3 Implementation notes

Here we point out some details about the implementation:

- Matrices distribution was handled by evenly distributing rows among processes. If the number of rows could not be divided exactly by n_{procs}, the remainder rows were assigned to first processes. In any case all rows assigned to a process were adjacent in the global matrices.
- 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 threads (openMP was used to speed up the initialization).
- Given the nature of communications between processes, they were implemented using the collective function `MPI_Allgatherv`.
- When using BLAS and cuBLAS, we accumulated the computed entries of C_{loc} directly on the memory allocated for C_{loc} (without using an auxiliary matrix). To do that, at each iteration of the algorithm we passed to `dgemm` the pointer to the first element of the block of C_{loc} to be computed as pointer to the result matrix and the number of columns of C as the stride argument.
- Considering that cuBLAS `dgemm` function assumes column-major order, we decided to avoid transpositions by computing on each process $B^T x A^T$.

1.4 Results

The following are the scaling results obtained on Leonardo's partition ... .

MPI processes were distributed one per socket (i.e two processes per node) for non GPU-accelerated programs and one per device for GPU-accelerated programs (i.e. four processes per node).

We measured time for initialization of matrices, communications and computation. Times for communication and computation were computed as the sum of times for communications and computations of all n_{procs} iterations of the algorithm.

Notice that in the program with cuBLAS we included in computation time the time for all data copies between host and device, since we were primarily interested in measuring gain in performance when accelerating programs with GPU.

2 OpenACC assignment

2.1 Introduction

In this second assignment we had to take a serial code implementing Laplace equation by Jacobi method, make it parallel with MPI, accelerate it using openACC, and study its scalability. The Jacobi evolution is performed on a square grid of cells with static boundary conditions.

We implemented both a CUDA-aware MPI and a CUDA-anaware MPI version of the program.

2.2 Parallel algorithm

CUDA-unaware version of the program:

1. Each process allocates its own portion of the grid (divided as blocks of rows), with an additional contour of cells as boundaries. Also, an identical auxiliary grid to store updated cells state is allocated.
2. Each process initializes its own cells, lateral boundaries and, only for first and last processes, upper and lower boundaries.
3. Both main and auxiliary grids are moved to device.
4. For the required number of iterations, repeat the following:
    - each process communicates to the bordering processes the corresponding neighbor rows (i.e. the first row to the upper process and the last one to the lower process) from host to host;
    - received boundaries are updated on device;
    - each process evolves its own cells on device on the auxiliary grid and then copies the updated system state to the main one;
    - updated border cells are copied back to host.
5. The final grid is copied back to host.

CUDA-aware version of the program:

1. Same as in CUDA-unaware.
2. Same as in CUDA-unaware.
3. Same as in CUDA-unaware.
4. For the required number of iterations, repeat the following:
    - each process communicates to the bordering processes the corresponding neighbor rows (i.e. the first row to the upper process and the last one to the lower process) directly from device to device;
    - each process evolves its own cells on device on the auxiliary grid and then copies the updated state to the main one.
5. Same as in CUDA-unaware.

2.3 Implementation notes

Here we point out some details about the implementation (we call n_{procs} the number of MPI processes):

- Grid distribution was handled by evenly distributing rows among processes. If the number of rows could not be divided exactly by n_{procs}, the remainder rows were assigned to first processes. In any case all rows assigned to a process were adjacent in the global grid.

- To speed up the grid initialization we used openMP.
- Boundaries communications between processes were performed by "exchanging" bordering rows between neighbor processes. To do that we used `MPI_Sendrecv()`, and `MPI_PROC_NULL` to make code more readable.
- Since pointer swapping is not supported in openACC, we had to copy all data from a grid to the other to update the system state. Also, we could not alternate main and auxiliary grid in even and odd iterations, since openACC directives cannot be put inside if statement. However, notice that performing matrix copy on GPU is an extremely fast operation (especially using the `independent` clause, as we did).
- Data movements were managed by placing an openACC data region around the for loop performing the evolution of the grid, and using clauses `copy` for the main grid and `copyin` for the 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.

2.4 Results

The following are the scaling results obtained on Leonardo's partition ... .

MPI processes were distributed one per device (i.e. four processes per node).

We measured time for initialization of grids, communications and computation. Times for communication and computation were computed as the average of times for communications and computations over all iterations of the algorithm, both averaged over all processes.

In this case, differently from the first assignment, we decided to include data transfers between host and device in communication time, since we were more interested in the difference between MPI aware and not aware communications.

3 MPI Remote Memory Access assignment

3.1 Introduction

In this last assignment we took the same serial code as in the second one, and made it parallel using MPI Remote Memory Access paradigm.

The code was further parallelized using openMP.

3.2 Parallel algorithm

The algorithm is the following:

1. Each process allocates its own portion of the grid (divided as blocks of rows), with additional columns as lateral boundaries, and also an additional array to store the upper and lower boundaries. Also, identical auxiliary grid and auxiliary border array to store updated cells state are allocated.

2. Each process initializes its own cells, lateral boundaries and, only for first and last processes, upper and lower boundaries.

3. For the required number of iterations, repeat the following:

   - each process opens a window on its border arrays to allow (only) neighbor processes to write directly to it;
   - each process writes directly to the bordering processes' border arrays the corresponding neighbor rows (i.e. the first row to the upper process and the last one to the lower process);
   - each process closes the opened window;
   - each process evolves its own cells on the auxiliary grid and then swaps main and auxiliary grids.

3.3 Implementation notes

Here we point out some details about the implementation (we call n_{procs} the number of MPI processes):

- Grid distribution was handled exactly as in the previous assignment (see ...). The only difference is in the fact that here upper and lower boundaries were separated from the actual grid, to make communications easier.
- To speed up grid and boundaries initialization and grid evolution we used openMP.
- For RMA operations we used the `MPI_Put` function for direct writing.
- To make communications more efficient we did the following:

  1. we created for each process an `MPI_Group` containing only its neighbors, and exposed the window exclusively to them;
  2. we used `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete` and `MPI_Win_wait` for exposing and accessing the windows; in this way, each process could proceed with computation once its window was written and the buffers to target processes were sent, without having to wait for the writing operation to be actually completed on the targets;
  3. the boundary array was allocated contextually to the window using `MPI_Win_allocate`.

3.4 Results

The following are the scaling results obtained on Leonardo's partition ... .

MPI processes were distributed one per socket, i.e. two processes per node.

Times were measured as in the previous assignment (see ...).

References