Advanced HPC - report

Gabriele Codega

October 2024

1 Introduction

This brief report contains some notes and details about the final project for the course in Advanced HPC, held at University of Trieste.

There will be one section per exercise, namely distributed matrix multiplication and the solution of the heat equation via Jacobi iterations.

All results are obtained by running the code on Leonardo at CINECA.

2 Matrix multiplication

2.1 Problem

The goal was to implement a distributed matrix multiplication using MPI. The distribution of data among processes happens through a 1D decomposition of the domain, where matrices are split in such a way that each process stores a certain number of rows of the whole matrix.

Given two matrices A and B of size $n \times n$, each process stores a portion A_{loc} and B_{loc} of the full matrices, both of size $nloc \times n$. Since we wish to compute the full product $AB = C$, processes need to communicate their B_{loc} to each other.

Each entry of C is given by the product of a row in A and a column in B, but because of how we split the matrices, only a portion of each column of B is available to each process. Then, we set up a communication pattern in such a way that at every communication, each process gets a chunk of the other processes' B_{loc} and stores it in a buffer, that we call B_{loc}, whose size is $n \times m$.

To be more specific, if *npes* is the number of processes, we do a total of *npes* communications, where each process sends m columns of B_{loc} to the others. In practice this is achieved through a call - or rather *npes* calls in total - to `MPI_Allgatherv` (to account for unequal workloads among processes). Since the data needs to be contiguous for sending, before each communication, processes copy the data to be sent from B_{loc} into a contiguous memory buffer (of size $nloc \times m$.

The result is that after each communication every process can compute $A_{loc}B_{col}$, resulting in a chunk of C of size $nloc \times m$.

The process is summarised in the following pseudo-code

```
for i = 1...npes
    Figure out how many columns need to be sent;
    Create a block from B_loc;
    MPI_Allgatherv(...);
    C_block = matmul(A_loc, B_col);
end
```

Here, the `matmul` function refers to a general matrix multiplication operation. In fact, we test the performance of this code for three different matrix multiplication routines:

- A naive, homemade multiplication routine
- `cblas_dgemm`, implemented in Intel MKL
- `cublasDgemm`, which exploits GPUs

2.2 Results

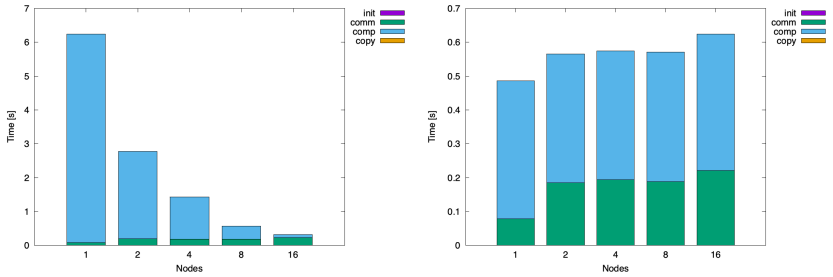
All the code was ran on CINECA Leonardo. With the idea of getting the best available computational resources for a given code, the CPU tests were carried out on the DCGP partition and the GPU tests were carried out on the BOOST partition.

In DCGP, each node is equipped with 2 Intel Xeon Platinum 8480p, 2.00 GHz processors, for a total of 112 physical cores per node. In BOOST, each node is equipped with 1 Intel Xeon Platinum 8358, 2.60GHz CPU, for a total of 32 cores, and 4 NVIDIA Ampere GPUs. On BOOST, the 32 cores were exploited to speed up the initialisation of data on CPU.

As for the software, CPU tests employed Intel oneApi toolkit, which provides the icc compiler, the mpiicc wrapper for OpenMPI and Intel MKL for linear algebra. GPU tests, on the other hand, employed NVIDIA's NVHPC SKD, which ships with the nvcc compiler, compatible with CUDA.

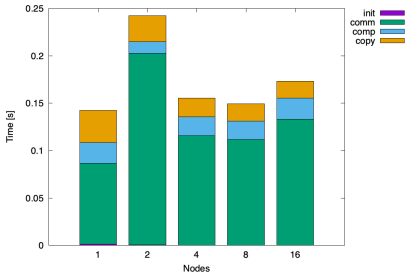
The following figures show boxplots for execution time for different numbers of nodes. The time has been split into time for initialisation, time for communication, time for computation and time for host-device memory copy. All times are averages over the processes. Note that for CPU tests, each node was assigned 2 processes (one per CPU), while for GPU tests, each node was assigned 4 processes (one per GPU).

The plots show that for the smaller matrix the naive implementation has some good scaling properties with the number of nodes. Indeed, the computation time is generally much larger than initialisation and communication. On the contrary, for the blas and cublas, the matrix is still too small to see a good scaling with the number of nodes. The time required for computation does not change appreciably with the number of nodes and the time for communications is a good fraction of the total execution time. Overall we can observe that GPU code is much faster than the others.



(a) Naive

(b) dgemm_cblas



(c) cublasDgemm

Figure 1: Boxplots for the three multiplication routines and a matrix of size 5000×5000.

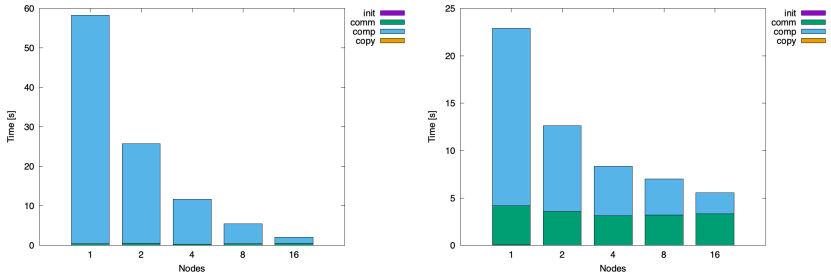
For the larger matrix we can see that blas and cublas computation times also decrease with the number of nodes. The time for communications and for host-device memory copies is almost constant (up to some oscillations that could be attributed to the network). Notice that here the matrix for the naive algorithm is 16 times smaller than the others. This is because the algorithm is so slow that using the same size matrix would be an unnecessary waste of resources.

3 Jacobi

3.1 Problem

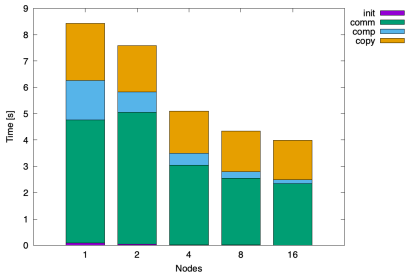
The goal here is to parallelise a pre-existing code for the solution of the heat equation via Jacobi iterations. The solution to the equation is represented as a matrix, whose entries are the value of the field at a given spatial location. The iterations consist in updating the values of the matrix for the step $t + 1$ based on the values at step t. In particular, to update each cell, we need the values in its four neighbouring cells.

This update scheme, when implemented in distributed memory requires each process to communicate with its neighbours at each iteration. To see why this is the case, consider the situation where each process owns *nloc* rows of the full matrix (which is what is done in practice here), and consider process p. Process



(a) Naive

(b) dgemm_cblas



(c) cublasDgemm

Figure 2: Boxplots for the three multiplication routines and a large matrix. For blas and cublas the size is 40000×40000, for naive it is 10000×10000.

p wishes to update the value of a cell in row 0. Since the general update step for cell in row i requires values in rows $i-1$, i and $i+1$, process p would need a value store in row -1, which in practice is a value stored in row $nloc-1$ (we are counting from zero) by process $p-1$. A symmetric case holds when process p tries to update row $nloc-1$.

Each iteration thus requires a communication between each process and its two neighbours, where the process sends the state of its first and last rows and gathers the information about the first and last rows of the next and previous processes, respectively.

In practice, each process allocates two extra rows to accommodate data from neighbours and the communication is implemented as two calls to `MPI_Sendrcv`. In the first call, each process sends its first row to the preceding process and receives the first row of the following process. In the second call, each process sends its last row to the following process and receives the last row from the preceding. This schema works because the first and last processes have `MPI_PROC_NULL` as one of their neighbours, and communications with the null process result in a `no_op`.

At the end of the iterative process, the data is written on a binary file in parallel with MPI IO.

```
Initialise data;
```

```
...
for it=1...max_iterations

    MPI_Sendrcv: send first row to previous, receive first row from next;
    MPI_Sendrcv: send last row to next, receive last row from previous;

    update matrix;

    exchange new and old matrix;
end
```

For this algorithm we compare a parallel CPU version, that uses MPI and OpenMP multithreading, a GPU version that uses OpenACC to offload computations to accelerators and employs cuda-aware MPI, and a GPU version that uses MPI Remote Memory Access (RMA) to perform one-sided communications (also cuda-aware).

For the OpenACC version, the iterations are enclosed in a data region: the initial state of matrices is copied on the device at the beginning, while at the end of the last iteration the final state of the matrix is copied back in the host memory. Since we want communications to be cuda-aware, the calls to MPI communications routines are enclosed in a `host_data use_device(...)` region, where we require the buffers used for communication to be those stored on device memory.

For the RMA version, the communications happen in a different way. Indeed, since the idea here is to use one-sided communications, instead of having processes send their first and last rows, each process opens a memory window on said rows so that the neighbours can directly get the values they need.

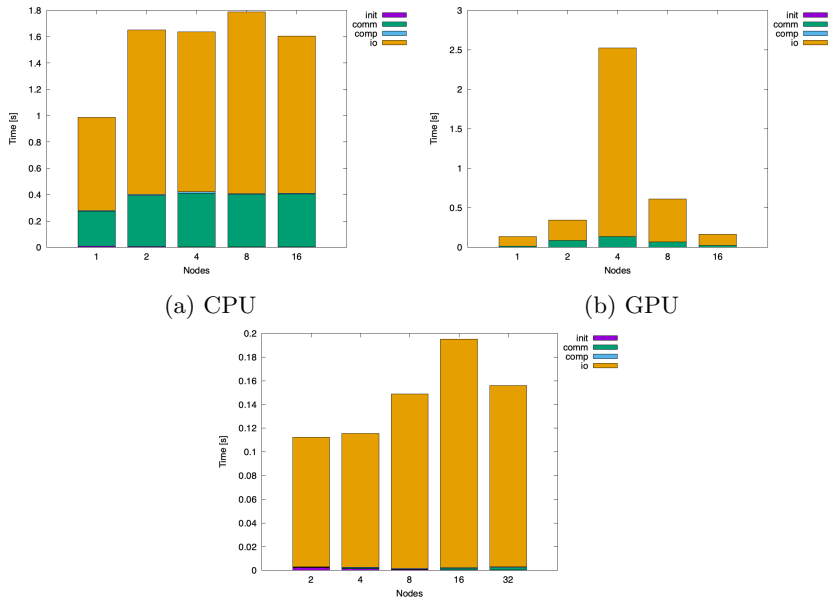
A schematic version of the algorithm is reported here.

```
Create windows and attach to the first and last rows;
for it=1...max_iterations
    MPI_Fence(top_window);
    MPI_Fence(bot_window);
    MPI_Get values in top_window of next process;
    MPI_Get values in bot_window of previous process;
    MPI_Fence(top_window);
    MPI_Fence(bot_window);

    update matrix;

    exchange new and old matrix;
end
```

3.2 Results

As for the previous exercise, all tests are run on CINECA Leonardo, on the DCGP and BOOST partitions for CPU and GPU respectively.

Boxplots for time of execution with different numbers of nodes are presented for two different sizes of the domain. Times refer to initialisation, communication, computation and i/o operations (writing the solution to file at the end). The number of iterations is set to 10 in all cases.



(a) CPU

(b) GPU

(c) GPU+RMA

Figure 3: Boxplots for the three Jacobi evolutions. The domain size is 1200×1200 and the number of iterations is 10.

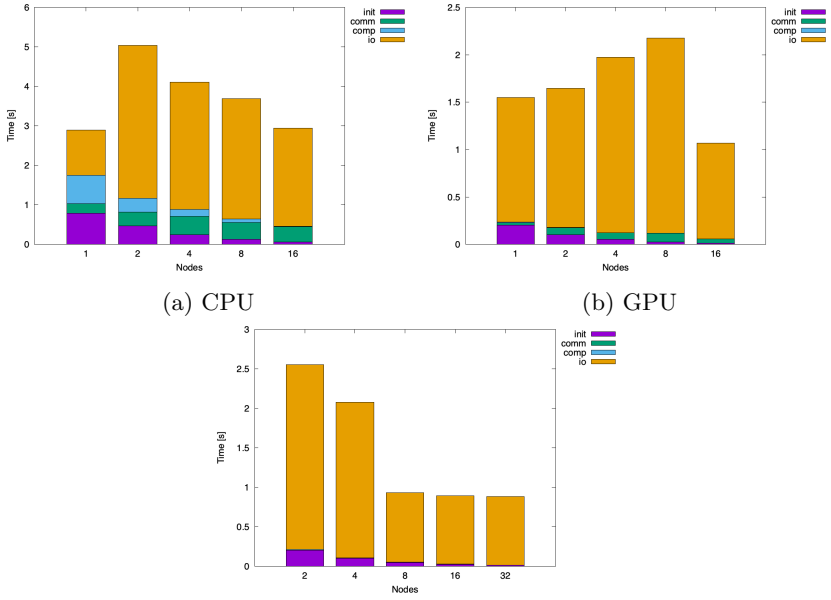
From the plots we can see that for the smallest domain size, all three implementations spend the most time in i/o. The CPU implementation also spends considerable time in communication, and from the GPU plot we can see that there may have been changes in the state of the network across different experiments.

The plots for the largest domain size instead show more details about the scaling. In particular, we can see that for the CPU code, both initialisation and computation times decrease with increasing number of nodes, while communication time slightly increases. It is also clear that i/o time grows significantly from 1 to 2 nodes, possibly because of internode communications and more complex synchronisation. After that sudden increase, it slowly decreases for increasing number of nodes.

Something similar is observable for GPU and GPU+RMA. The initialisation in these cases still happens on CPU, so it can be seen that the time decreases with increasing number of nodes, while the computations now are run on GPU, and thus are so fast that can't be seen at the scale of these plots. For the GPU case we can also see that communication time is larger for larger numbers of

nodes and that i/o time increases with number of nodes until suddenly dropping for 16 nodes. This could be attributable to the state of the network. In the GPU+RMA case, communication time is also not noticeable at this scale, while i/o time decreases significantly until reaching a plateau for 4 nodes or more.

Finally, Figure 5 shows a comparison between GPU and GPU+RMA. Since the two codes are identical in everything but the communication scheme, we can see that initialisation and computation times are indeed equal in the two cases. What is important here is that the communication times, in orange, are significantly smaller for GPU+RMA, proving that for this particular application one-sided MPI communications can be beneficial.



(a) CPU

(b) GPU

(c) GPU+RMA

Figure 4: Boxplots for the three Jacobi evolutions. The domain size is 12000 × 12000 and the number of iterations is 10.

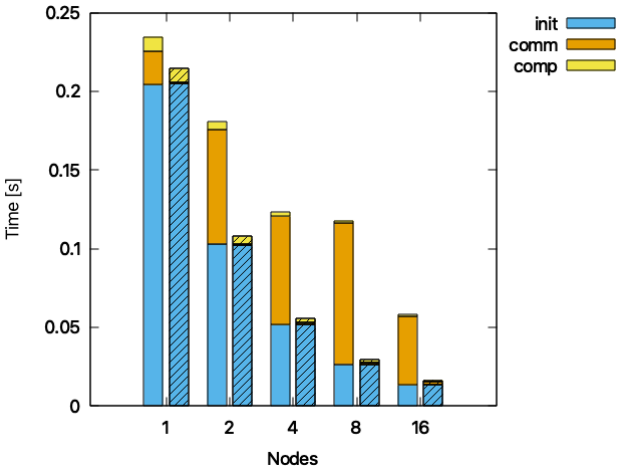


Figure 5: Comparison of times for the standard two-sided communications and the Remote Memory Access communications. Solid bars are two-sided, bars with pattern are RMA. Domain size is 12000×12000.