Using MPI to efficiently distribute GEMM computations

To set up the environment, you have to execute in a terminal the command
source *init.sh*

1 Introduction

1.1 Message Passing Interface (MPI)

The MPI standard emerged in the beginning of the 90s to consolidate a frame to use network-connected devices. Those architectures were prominent as this time as they allowed to draw more computing power from multiple single-core nodes. MPI is implemented in several libraries - Openmpi [1], intelMPI [2] and MPICH [3] to name a few - and new implementations still get created to further enhance the standard, especially towards large-scale architectures of tens of thousands of multi-core nodes.

During this class, we will use SimGrid [4] which is an emulator of MPI. This emulator allows the simulation of the network to the extend that you are not limited by the machines available in the classroom. We will focus on simpler routines such as `MPI_Ssend`, `MPI_Recv`, `MPI_Bcast`, `MPI_Issend`, `MPI_Irecv` and core concepts of MPI such as Communicators.

The compilation of your MPI application is handled in the provided `Makefile` through the `smpicc` wrapper. In order to launch an MPI application, one simply needs to call `smpirun /path/to/mpi/app` with eventual arguments. All of this tinkering is made easier thanks to scripts handed out with this subject.

1.2 Distributed memory General Matrix Multiplication (GEMM)

The sequential GEMM routine is **a principal routine** in any scientific applications. Its single-core implementation is available through `BLAS` libraries as an efficient and necessary building block of more complex operations as found, for instance, in `LAPACK` libraries. Despite being a simpler mathematical operation, **its numerical implementation is not trivial** as it is tied with the hardware used to run the floating-point instructions.

[1] `https://github.com/open-mpi/ompi`
[2] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.htm`
[3] `https://github.com/pmodels/mpich`
[4] `https://framagit.org/simgrid/simgrid`

1.2.1 2D Block-Cyclic distribution of dense matrices

We are interested in measuring the performances of various GEMM algorithms in a distributed memory setting. We do not focus on multicore applications and we will only use one core per node (`EXPORT {MKL,OMP}_NUM_THREADS=1`). We restrict ourselves to the routine applied to **dense matrices** of single precision floats (`float`). Matrices will be distributed over the network through an usual pattern : the **2D-Block Cyclic** (2DBC) way.

For convenience, each block in a matrix will be a square block of dimension b and we will assume each dimension of a matrix is divisible by b. The 2DBC pattern means that a given $A_{i,j}$ block will be stored on the memory of the node $q * mod(i,p) + mod(j,q)$ if the nodes are arranged locally on a $p \times q$ grid. Each node can be described by its position $r \times c$.

You will find an implementation of this representation in the source files in `./src/dsmat.h`. Figure 1 describes the 2DBC pattern for a 40×50 matrix with a blocking of 10 stored on a $p \times q = 3 \times 2$ grid of 6 nodes.

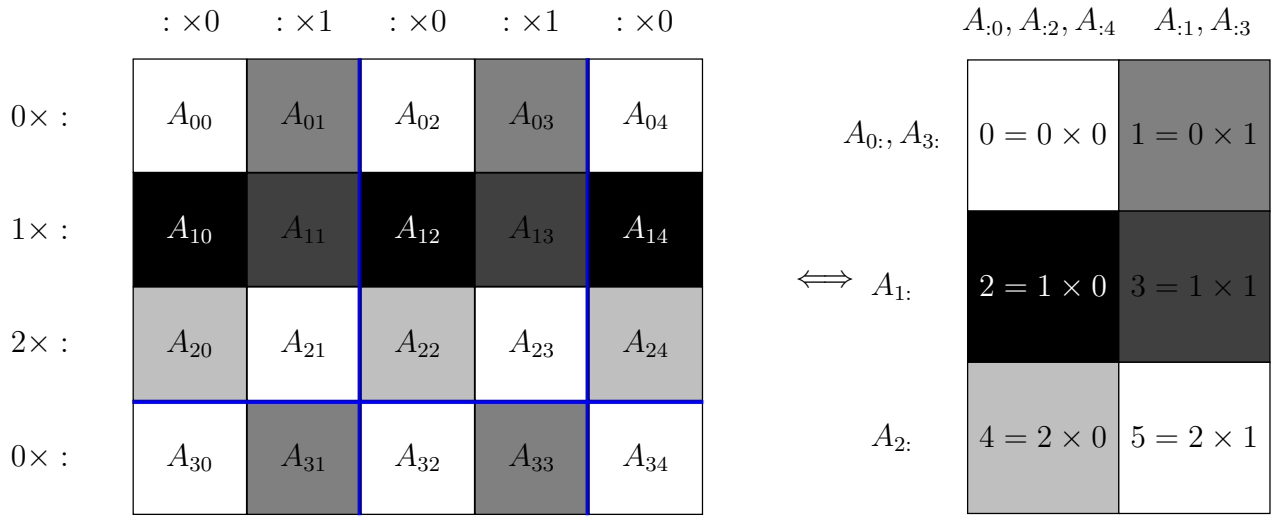


Figure 1: A split matrix distributed on nodes $\iff$ the grid of nodes and their affected column/row combination of blocks

1.2.2 Some notations

GEMM is standardized as the following operation

$$C = \alpha op_A(A) \times op_B(B) + \beta C$$

where op are either identity or transposition, α and β scalar and A, B, C matrices. We will focus on a simplification of this operation by setting $\alpha = 1, \beta = 0, op_A = op_B = Id$ hence we will compute

$C = A \times B$. C is a $m \times n$ matrix and A and B share a dimension k.

Using the 2DBC pattern, GEMM can be described as **a set of block-wise operations** that can be identified with the (i, j, l) triplet : efficiently executing GEMM using distributed memories is a matter of assigning, scheduling and executing the computations of $C_{i,j} = C_{i,j} + A_{i,l} \times B_{l,j},\ \forall i, j, l$ over a given number of nodes. The algorithms you will implement are based on the following **static scheduling**: each node will compute the blocks it owns in the order given by increasing value of l. The `sgemm` routine is provided by the Maths Kernel Library (MKL) BLAS library : we assume such a routine is as efficient as the hardware allows.

2 Exercises

The section 3 goes through some declarations of `src/utils.h` and `src/dsmat.h` that could prove useful to tackle the following exercises.

Figure 3 shows the transmissions of blocks of $A_{:,l}$ and $B_{l,:}$ for a given value of $l(= 2)$. You can observe that the blocks of A are transmitted row-wise and the blocks of B column-wise.

The execution trace produced by the algorithms you will be writing can be visualized. By default, they are named `smpi_simgrid.trace`. These traces can be opened through Vite [5].

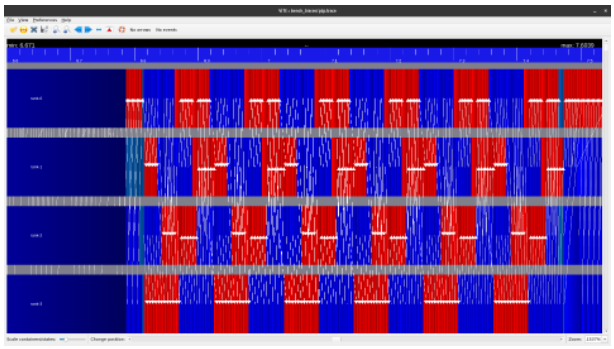


Figure 2: Example of a trace opened with Vite

2.1 Blocking peer-to-peer communications (TP)

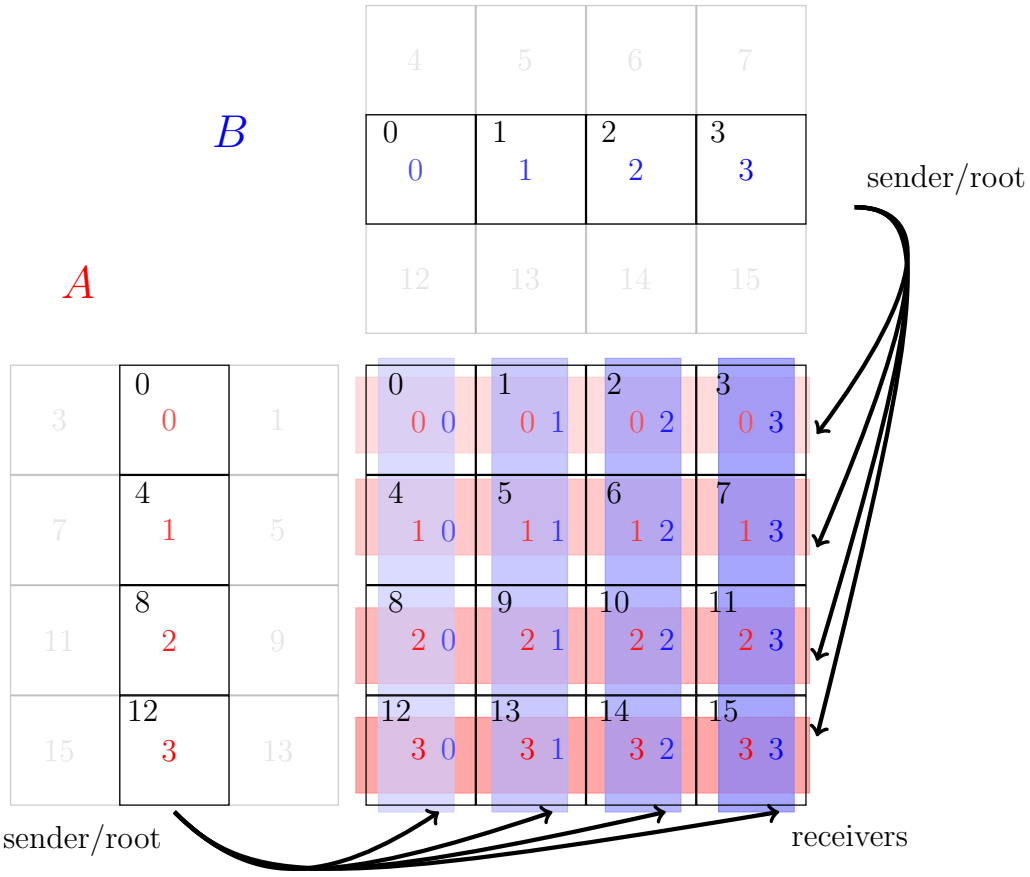
Using the routines `MPI_Recv` and `MPI_Ssend`, modify the functions `p2p_transmit_A` and `p2p_transmit_B` in the source `./src/ex1.c` to obtain an algorithm that matches with the pseudocode given in Algorithm 1.

You should measure the performance of your algorithm on several matrices, eventually on several grid sizes.

By default, if not specified in the command line, **the 3 sizes m, n, k are equal to 20**.

[5] `https://gitlab.inria.fr/solverstack/vite`

Figure 3: Communication patterns on a 4×4 grid for the transmission of one column of A and one row of B for step $l = 2$



Here is an example command-line to test your implementation[6]

```
n7>  smpirun -platform platforms/cluster_crossbar.xml -hostfile hostfiles/cluster_hostfile.txt -np 4 ./build/bin/main -p 2
-q 2 -n 10 -b 5 --algorithm p2p --niter 2 -c -v
```

and with the genereration of the trace file

```
n7>  smpirun -platform platforms/cluster_crossbar.xml -hostfile hostfiles/cluster_hostfile.txt -np 4 -trace ./build/bin/main
-p 2 -q 2 -n 10 -b 5 --algorithm p2p --niter 2 -c -v
```

[6]be careful with the copy/paste of the command: you should copy/paste the two lines separately and also check for the underscores '_'

2.2 Blocking collective communications (TP)

Using the routine `MPI_Bcast`, modify the functions `bcast_A` and `bcast_B` in the source `./src/ex2.c` to obtain an algorithm that matches with the pseudocode given in 2. As collective communications operate in communicators, `MPI_COMM_WORLD` has been split through the `MPI_Comm_split` routine in order for each node to know a communicator of its row neighbours and a communicator of its column neighbours.

You should measure the performance of your algorithm on several matrices, eventually on several grid sizes.

Here is an example command-line to test your implementation

```
n7>  smpirun -platform platforms/cluster_crossbar.xml -hostfile hostfiles/cluster_hostfile.txt -np 4 ./build/bin/main -p 2
-q 2 -n 10 -b 5 --algorithm bcast --niter 2 -c -v
```

2.3 Non-blocking peer-to-peer communications (homework)

`MPI_{Ssend,Recv}` return once the communications have been executed on their respective nodes. In order to allow for computation and communication to overlap, we may use non-blocking communication patterns. These patterns require the use of `MPI_Request`s to wait or test the execution of a given communication.

Using the routines `MPI_Wait`, `MPI_Irecv` and `MPI_Isend`, modify the functions `p2p_i_transmit_A`, `p2p_i_transmit_B` and `p2p_i_wait_AB` in the source `./src/ex3.c` to obtain an algorithm that matches with the pseudocode given in Algorithm 3.

You should measure the performance of your algorithm on several matrices, eventually on several grid sizes. Try different `lookahead` values.

Here is an example command-line to test your implementation

```
n7>  smpirun -platform platforms/cluster_crossbar.xml -hostfile hostfiles/cluster_hostfile.txt -np 4 ./build/bin/main -p 2
-q 2 -n 10 -b 5 --algorithm p2p-i-la --lookahead 2 --niter 2 -c -v
```

2.4 ModIA (2 person job)

As you have implemented several variations of a distributed GEMM algorithm, you are able to benchmark them.

The script `bench.sh` produces execution speed measures stores in `bench.csv`[7] and you can use the given `example.gnuplot` file to produce graphs from the csv.

```
n7> gnuplot -e "filename='bench.csv'" example.gnuplot
```

> You can, of course, write other scripts according to your plan(s) of experiments and also use your own visualization solutions (python?) to display the results.

The benchmark does not test scalability as handed out to you: a number of nodes is chosen -p and q - and the problem - the values of m, n, k - is changed to observe a comparison of these algorithms.

Your goal is to assess how weakly (**OR** strongly) scalable the different algorithms are and whether their different approach (collective operations, non blocking communications) have a strong impact on performance.

In order to meaningfully assess the scalability of the algorithms, you should carefully chose values of m, n and k as (**OR** while) you increase p, q. The choice of these values can be motivated by analyzing the distribution and the complexity of the computation over the machines. Your analysis can be completed by observing `VITE` traces to ensure the performance and behavior of the algorithms are coherent.

This analysis (how and what you plan on experimenting, results of your experiments, conclusions from your results) should be reported in a 15 pages document (max).

NOTE I: the source file (`main.c`) is able to read the 3 dimensions of the problem rather than decide $m = n = k$ as we did during the hands-on. This should ease the choice of m, n, k to match with p, q[8].

NOTE II: with Simgrid, you can easily change your computational platform. It could be interesting to compare the behavior of the three implementations against the platforms (see directory *platforms*).

[7] Read it carefully to understand what are the results it produces

[8] Be careful: because for the hands-on, $m = n = k$, perhaps, if you now take different values, there will be some errors with your code. You should check the dimensions carefully for each operation, because now these dimensions can be different

3 Documentation

This section describes the two mini-libraries that can be used in this class to implement the algorithms in a simpler fashion.

3.1 Dense matrices mini-library

`./src/dsmat.h` provides the type `Matrix` to describe a matrix of any size. We assume a matrix is subdivized into blocks of square size `b` and that a given matrix is of size `mb*b` $\times$ `nb*b`. Assuming a matrix X has been instanciated in `X`, the sub-matrix $X_{i,j}$ can be accessed through the `blocks` member of X through `X.blocks[i][j]`.

Each block is populated with its content (`c` : assuming x is a block of size b, $x_{i,j}$ can be accessed with `x.c[b*i+j]`), who owns it (`owner`, the (MPI) rank of the owner), as well as the position of the block in the logical grid of owners (assuming the owners are arranged in a $p \times q$ grid, we require `x.owner == q*x.row + x.col`). A `MPI_Request` is associated with each block, in case the block is transmitted via a non-blocking communication.

Routines are provided to fill, scale, copy, ... blocks or complete matrices but you do not have to use them to complete the exercises. The computation as well as the memory management is handled in those routines.

3.2 Utilities mini-library (distributed memory part)

Most of the routines in `./src/utils.h` are not useful to carry on with the class. The distributed-memory routines should, however, make the exercises easier to implement.

The exercises require you, among other things, to identify the owning node or the location of a node in a grid. Each node has a MPI rank on the `MPI_COMM_WORLD` communicator that can be obtained through the function `MPI_Comm_rank`.

The position of a node in a $p \times q$ logical grid can be extracted using its MPI rank me through either `node_coordinates` or `node_coordinates_2i`. If one wants to retrieve the MPI rank of a node at position (i, j) in the logical grid of $p \times q$ nodes, the function `get_node` returns such an information.

4 Algorithms

Algorithm 1 distributed GEMM algorithm inspired by SUMMA without collective communications

for $l = 1, ..., k$ **do**
  for $i = 1, ..., m$ **do**
    transmit $A_{i,l}$ to the nodes in my row
  end for
  for $j = 1, ..., n$ **do**
    transmit $B_{l,j}$ to the nodes in my column
  end for
  for $i = 1, ..., m; j = 1, ..., n$ **do**
    compute $A_{i,l} * B_{l,j}$ contributing to $C_{i,j}$
  end for
end for

Algorithm 2 distributed GEMM algorithm inspired by SUMMA with collective communications

for $l = 1, ..., k$ **do**
  for $i = 1, ..., m$ **do**
    broadcast $A_{i,l}$ in my row
  end for
  for $j = 1, ..., n$ **do**
    broadcast $B_{l,j}$ in my column
  end for
  for $i = 1, ..., m; j = 1, ..., n$ **do**
    compute $A_{i,l} * B_{l,j}$ contributing to $C_{i,j}$
  end for
end for

Algorithm 3 distributed GEMM algorithm inspired by SUMMA with communication/computation overlap

for $l = 1, ..., lookahead$ **do**
  for $i = 1, ..., m$ **do**
    post $A_{i,l}$ transmission in its row
  end for
  for $j = 1, ..., n$ **do**
    post $B_{l,j}$ transmission in its column
  end for
end for
for $l = 1, ..., k$ **do**
  if $l + lookahead \leq k$ **then**
    for $i = 1, ..., m$ **do**
      post $A_{i,l+lookahead}$ transmission in its row
    end for
    for $j = 1, ..., n$ **do**
      post $B_{l+lookahead,j}$ transmission in its column
    end for
  end if
  wait for $A_{:,l}$ and $B_{l,:}$ to be transmitted
  for $i = 1, ..., m; j = 1, ..., m$ **do**
    compute $A_{i,l} * B_{l,j}$ contributing to $C_{i,j}$
  end for
end for