TP°1-2 : Using MPI to efficiently distribute GEMM computations

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, intelMPI and MPICH 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 OpenMPI 2.1 as it allows us to use some older yet useful software for post-mortem visualizations : Message Passing Environment (MPE). We will focus on simpler routines such as `MPI_(i)Recv`, `MPI_iSend`, `MPI_Ssend`, `MPI_Bcast` and core concepts of MPI such as Communicators.

The compilation of your MPI application is handled in the provided `Makefile` through the `mpicc` wrapper. In order to launch an MPI application, one simply needs to call `mpirun /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. It 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.

1.2.1 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.

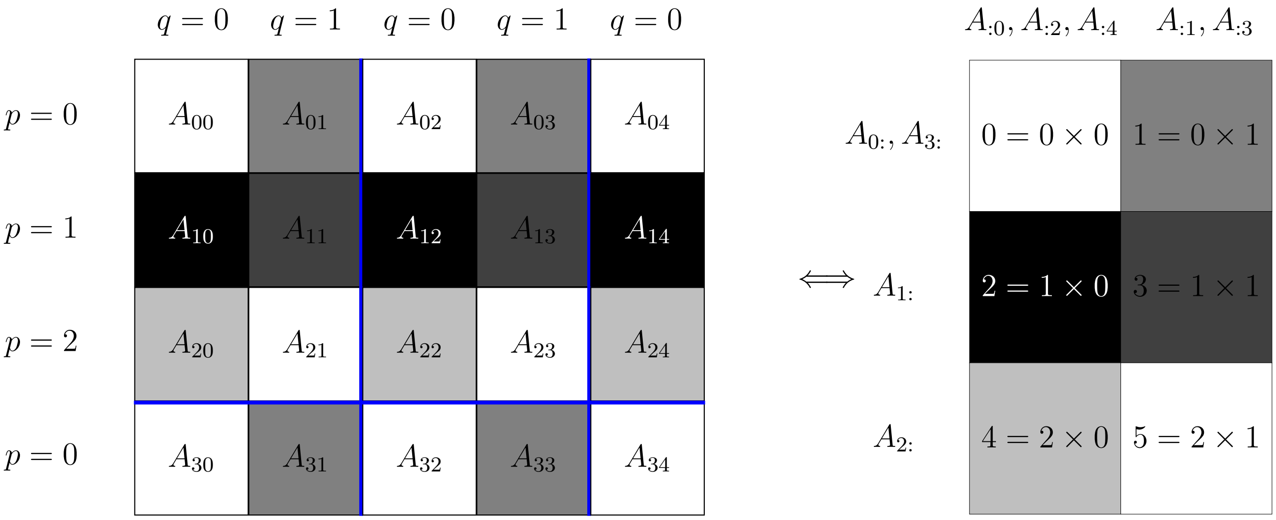

$q = 0$ $q = 1$ $q = 0$ $q = 1$ $q = 0$
$p = 0$
$p = 1$
$p = 2$
$p = 0$
A_{00} A_{01} A_{02} A_{03} A_{04}
A_{10} A_{11} A_{12} A_{13} A_{14}
A_{20} A_{21} A_{22} A_{23} A_{24}
A_{30} A_{31} A_{32} A_{33} A_{34}
$\iff$
$A_{:0}, A_{:2}, A_{:4}$ $A_{:1}, A_{:3}$
$A_{0:}, A_{3:}$
$A_{1:}$
$A_{2:}$
$0 = 0 \times 0$ $1 = 0 \times 1$
$2 = 1 \times 0$ $3 = 1 \times 1$
$4 = 2 \times 0$ $5 = 2 \times 1$

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 transposition or the identity, 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 $C = AB$. C is a $m \times n$ matrix and A and B share a dimension k. We decide that $m = n = k$ as a further simplification.

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 $C_{i,j} += A_{i,l} \times B_{l,j} \forall i, j, l$ 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.

1.3 How to run the practical class

☞ To avoid disturbing the network of your classroom, you are invited to coordinate with your teacher to launch larger tests. Specifically, you will want to test and check the correctness of your algorithms locally, on your working machine, by simulating many MPI nodes.

Be cautious about the size of the matrices you are multiplying :

- Each machine has 16GB of RAM which allow for a maximum of 3 square float-precision matrices of size 37,000 to be stored. You are not required to test your algorithms on matrices of size more than 16,000.
- GEMM requires $2mnk$ flops and one core on each machine in your classroom pulls about 90 Gflop/s with `sgemm` if the blocks are at least of size 512. You are not required to wait for gigantic matrices to compute.

1.3.1 Some useful command lines

You **should** `source utils.sh` to setup your environment and use the right `mpirun`.

To compile the source files you can use the following command - the produced executable is found in `./build/bin/`. You do not need to do all the exercises to compile your build : you can -should- work incrementally.

```
 n7> make
```

The program `./build/bin/main` takes into arguments the size of the grid ($p \times q$) as well as the dimension (n) and blocking (b) of the matrices - it expects b divides n. Along this practical class, you will implement various communication patterns for GEMM : each implementation can be summoned through the `--algorithm` option.

To test an algorithm locally, you will want to use the following command with a small amount of simulated nodes and a small workload. The `-c` option activates the check of the result and the `-v` option turns on verbosity. The `--lookahead` option is only meaningful to the `p2p-i-la` algorithm. You should make sure the size of the grid given to `mpirun` and to the `main` program match.

```
 n7> mpirun --oversubscribe --np [PxQ] build/bin/main -p [P] -q [Q] \
-n [n] -b [b] --algorithm [p2p|bcast|p2p-i-la --lookahead [la]] \
-v -c
```

➢ This command is launched in the script `check.sh` to test all your implementations.

In order to bench an algorithm with multiple nodes, you will have **to create machinefiles** handed out to `mpirun`. Usually, this would be handled by a workload manager such as `SLURM`. Machinefiles can be produced with the `add_ssh.sh` script. This script takes the number of the classroom as well as the number of machines you require to produce a working machinefile in the `hostfiles` directory. **It might require some input on your part to allow the recognition of machines you have never accessed.**

```
n7> bash add_ssh.sh 9 214
n7> cat hostfiles/214_9 # prints the selected machines
```

To actually bench an algorithm with multiple nodes, you will want to use the following command with a reasonable number of nodes (it is not necessary to exceed a 4×4 grid) and a sufficient workload. In order to fully use the nodes, you will want block large enough : $b = \{256, 512, 1024\}$ are interesting values to feed `BLAS-3` routines on the CPUs of N7. If you wanted to see how stable the performance of your algorithm is, you can use the `--niter` option to measure several iterations of your algorithm.

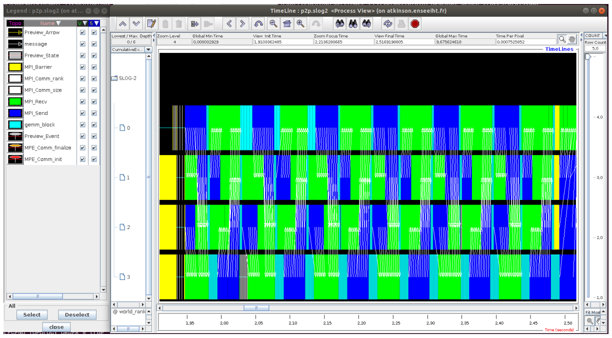
```
 n7> mpirun --hostfile [hostfile] --np [PxQ] build/bin/main -p [P] -q [Q] \
-n [n] -b [b] --algorithm [p2p|bcast|p2p-i-la --lookahead [la]] \
--niter [i] -c
```

➢ This command as well as the setting of the hostfile is launched through the script `bench.sh` to measure all your implementations.

Once you have tested your algorithm, traces are produced and stored by default in `Unknown.clog2`. You should rename this file in order to keep it. The traces can be visualized using the `jumpshot` program provided with MPE, as in Figure 2 - assuming `MPEDIR` as been
set to `/mnt/n7fs/ens/tp_guivarch/hpc2021/mpe2-2.4.9b`. Saves are made in the scripts `bench.sh` and `check.sh` to `bench_traces` and `check_traces`, respectively.

```
n7> mv Unknown.clog2 traces/<algo>.clog2
n7> $MPEDIR/build/bin/clog2TOslog2 traces/<algo>.clog2 &> /dev/null
n7> rm traces/<algo>.clog2
n7> $MPEDIR/build/bin/jumpshot traces/<algo>.slog2
```

Figure 2 – Example of a trace opened with MPE



Legend : p2p.slog2 (on at...

TimeLine : p2p.slog2 <Process View> (on atkinson.enseeiht.fr)

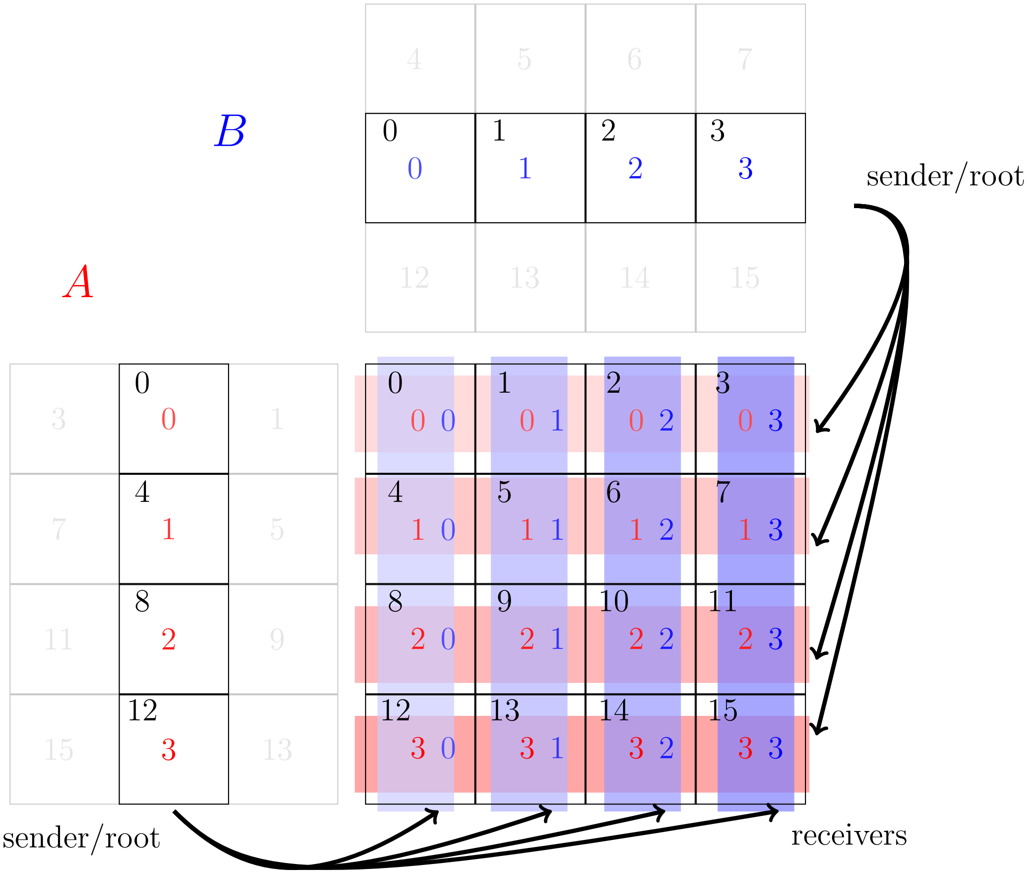
Lowest / Max. Depth: 0 / 6
Zoom Level: 4
Global Min Time: 0,000032929
View Init Time: 1,9103362485
Zoom Focus Time: 2,2136280665
View Final Time: 2,5169198805
Global Max Time: 9,675624618
Time Per Pixel: 0,0007525852
Row Count: 5,0

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. You can observe that the blocks of A are transmitted row-wise and the blocks of B column-wise.

Figure 3 – Communication patterns on a 4 × 4 grid for the transmission of one column of A and one row of B



2.1 Blocking peer-to-peer communications (TP1)

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.

Here is an example command-line to test your implementation

```
 n7> mpirun --oversubscribe --np 4 build/bin/main -p 2 -q 2 -n 10 -b 5 --algorithm p2p --niter
2 -c -v
```

2.2 Blocking collective communications (TP1)

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> mpirun --oversubscribe --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 (TP2)

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_Requests 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> mpirun --oversubscribe --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 Analysis

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 stored in bench.csv.

Modify this script in order to compare the algorithms. You should focus on observing either the weak scalability or the strong scalability : values of n should be chosen accordingly. You can view a graph of the execution speed of your experiments using

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

Does the performance of your implementations match with their behaviour as shown through MPE ?

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` × `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 asynchronously.

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 $j = 1, ..., m$ do
    transmit $B_{l,j}$ to the nodes in my column
  end for
  for $i = 1, ..., m; j = 1, ..., m$ 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 $j = 1, ..., m$ do
    broadcast $B_{l,j}$ in my column
  end for
  for $i = 1, ..., m; j = 1, ..., m$ 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
  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
  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
```