Université Grenoble Alpes, Grenoble INP, UFR IM^{2}AG

Master 1 Informatique and Master 1 MOSIG

UE Parallel Algorithms and Programming

Lab Matrix Product

2021

Introduction

In this lab, we we are going to work on the matrix multiplication problem $C = A \times B$. We are going to study different algorithms, that assume different virtual topologies for the processes. These algorithms are going to be implemented using MPI.

Exercise 1: Matrix product algorithm in a virtual ring topology

In this first exercise, we assume a virtual unidirectional ring topology, as was studied in the lecture about *Parallel Algorithms in Message-Passing systems*.

Write a MPI program that implements a matrix product in a virtual unidirectional ring topology.

- The algorithm to be implemented is described in Section 3 of the lecture notes.
- In this exercise, we will assume that initially all the data (matrices A and B) are stored on rank 0.
  - More specifically, it means that rank 0 is in charge of allocating and initializing matrices A and B. The blocks of A and B will then have to be distributed to all ranks
  - Note that you can use MPI collective operations to implement the data distribution.
- At the end of the execution, matrix C should be reconstructed on rank 0.

Advise: You can try implementing the Matrix-Vector multiplication first if you get confused with the computation of the indexes for the Matrix-Matrix multiplication.

Advise 2: Implement the Matrix-Matrix multiplication as a set of Matrix-Vector multiplications if you don't manage to implement the native Matrix-Matrix multiplication algorithm (You will still get points for that).

Provided code You are provided with an example of code that tests a matrix multiplication algorithm. The tested algorithm is sequential.

This code is provided to you because you can, if you want, reuse some of it to test your own code. It includes:

- A set of basic functions to manipulate matrices (alloc, init, compare, etc.). See files `utils.h` and `utils.c`.
- A Makefile to compile the code. The Makefile can take 3 optional parameters:
  - `RAND_INIT` (default value: 1): Initialize the matrices with random value.
  - `CHECK_CORRECT` (default value: 0): To test the result against a default sequential version
  - `EVAL_PERF` (default value: 1): To run a performance evaluation
  - Example:

```
make -B CHECK_CORRECT=1 EVAL_PERF=0
```

- An example of code that measures the performance and/or checks the correctness (depending on the compilation options) of a sequential matrix multiplication implementation. The provided code takes a single argument which is the size of the matrices to create (see `matrix_multiplication_mpi.c`). To be run as follows:

```
mpirun -n 1 ./matrix_multiplication_mpi.run 512
```

Exercise 2: Fox matrix product algorithm

Write a MPI program that implements a matrix product using Fox algorithm. Fox algorithm is described in Algorithm 1. Several of the primitives studied on the previous lab can be useful here.

Algorithm 1 Fox Algorithm

for k=1 to q in parallel **do**
  Broadcast of the k^{th} diagonal of A
  Local computation $C = C + A * B$
  Vertical shift of B

The first two iterations of the algorithm are presented Figures 1 and 2.

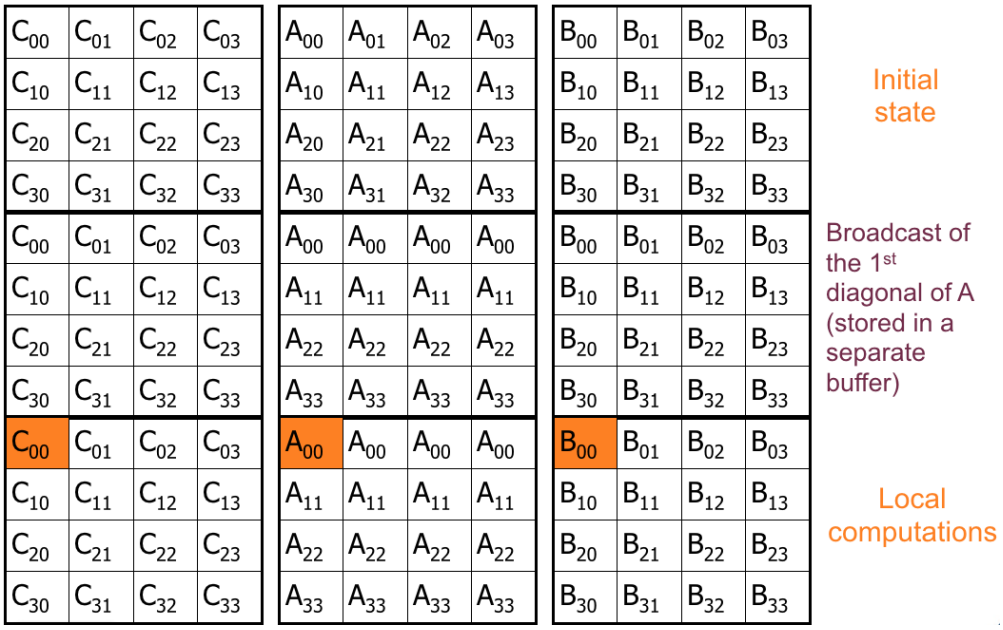

Figure 1: Step 1 of Fox algorithm

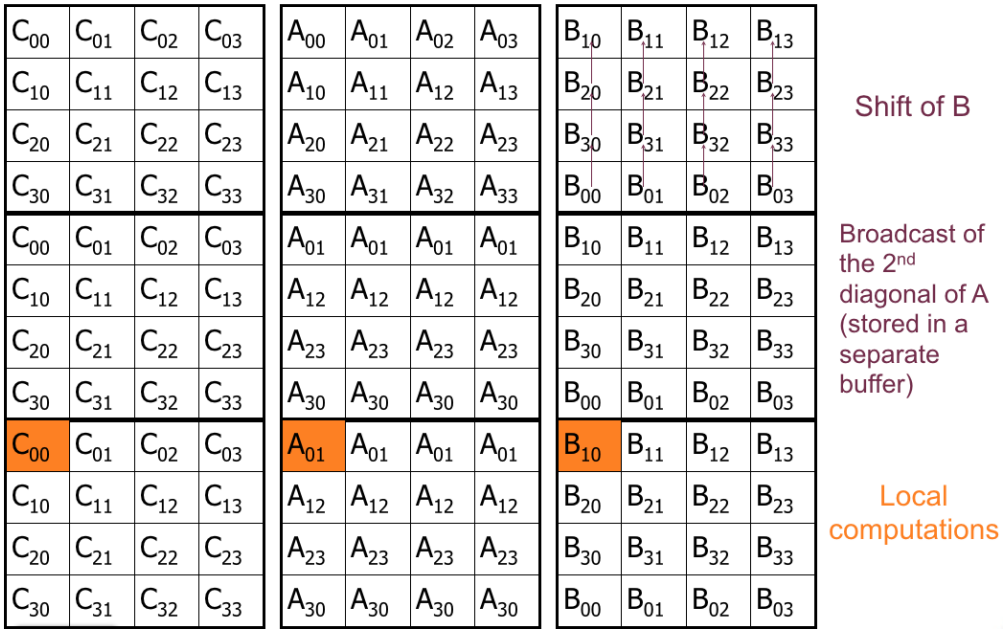

Figure 2: Step 2 of Fox algorithm

About data management (IMPORTANT): In this exercise, we are going to consider the following simplifying assumptions regarding data management:

- We will assume that the blocks of matrices A and B are already distributed over the processes in the grid. In practice, it means that at the beginning of the program, each MPI process is going to allocate

and initialize (with the values you want) its block of matrices A and B.

- We are going to ignore the last step of the algorithm, that is, gathering computed blocks of matrix C on rank 0.

Exercise 3: Data management

In this exercise, you are asked to implement a new version of the Fox algorithm, but this time implementing full data management. This means that:

- Rank 0 is in charge of allocating and initializing matrices A and B. The blocks of A and B will then have to be distributed in the grid.
- At the end of the program, the computed matrix C should be reconstructed on rank 0.

For this step, you might need to use the following MPI functions:

- `MPI_Type_vector()` and `MPI_Type_commit()`.
- `MPI_Scatterv()` and `MPI_Gatherv()`

Option 1: Performance evaluation

Based on the algorithms that you managed to implement, run a campaign of performance tests that illustrate the performance of your algorithms. **Your report should include:**

1. A description of the experiments you run
2. The result of the experiments
3. Your comments about these results

A few suggestions about the experiments:

- In a first step, you can consider running several MPI processes on the same machine.
- You can also consider using Virtual Machines in the Cloud to get a real distributed settings (see previous labs).

About the description on an experiment: For a performance evaluation to be meaningful, it should come with a description of the setup that is precise enough for the reader to know exactly what is evaluated. Among the necessary information that one should include in such a description, there is:

- A description of the hardware used for the experiments (How many processors? How many cores per processor? etc.)
- A description of the software configuration (How many MPI ranks? How is the mapping done on the physical resources?)
- A description of the methodology (What metric is considered? How many runs for a given configuration? How are results aggregated: median, average, ...? etc.)

Option 2: Cannon matrix product algorithm

Implement a matrix product using the Cannon algorithm, described in Algorithm 2. This algorithm starts with a redistribution of matrices A and B called *preskewing* (Figure 3). Then matrix multiplication is run as described in Figure 4. Finally the matrices are restored to their initial distribution during a phase called *postskewing*.

Algorithm 2 Cannon Algorithm

```
Participate to the preskewing of A
Participate to the preskewing of B
for k=1 to q do
    Local C = C + A*B
    Horizontal shift of A
    Vertical shift of B
Participate to the postskewing of A
Participate to the postskewing of B
```

A_{00}	A_{01}	A_{02}	A_{03}
A_{11}	A_{12}	A_{13}	A_{10}
A_{22}	A_{23}	A_{20}	A_{21}
A_{33}	A_{30}	A_{31}	A_{32}

B_{00}	B_{11}	B_{22}	B_{33}
B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}

Figure 3: After the preskewing of A and B

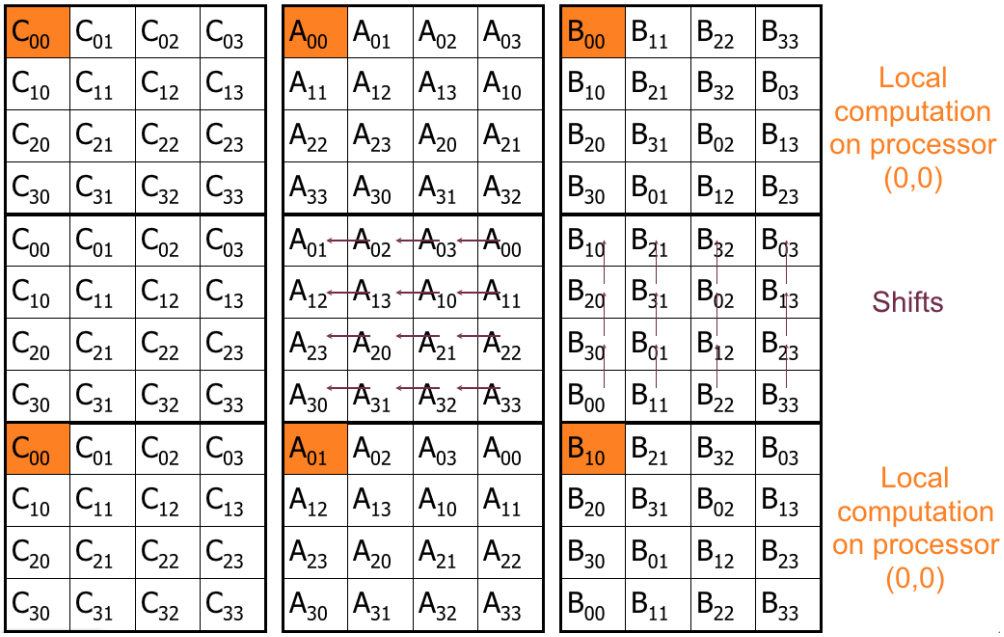

Figure 4: Steps of the Canon algorithm

Message Passing Interface - Quick Reference in C

Environmental

- `int MPI_Init (int *argc, char ***argv)` - Initialize MPI
- `int MPI_Finalize (void)` - Cleanup MPI

Basic communicators

- `int MPI_Comm_size (MPI_Comm comm, int *size)`
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`

Point-to-Point Communications

- `int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` - Send a message to one process.
- `int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)` - Receive a message from one process
- `MPI_SendRecv_replace()` - Send and receive a message using a single buffer

Collective Communications

- `int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)` - Send one message to all group members
- `int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` - Receive from all group members
- `int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` - Send separate messages to all group members

Communicators with Topology

- `int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)` - Create with cartesian topology
- `int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)` - Determine rank from cartesian coordinates
- `int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int *coords)` - Determine cartesian coordinates from rank
- `int MPI_Cart_shift (MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)` - Determine ranks for cartesian shift.
- `int MPI_Cart_sub (MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)` - Split into lower dimensional sub-grids

Constants

Datatypes: `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED_SHORT`, `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, `MPI_BYTE`, `MPI_PACKED`

Reserved Communicators: `MPI_COMM_WORLD`