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

Master 1 Informatique and Master 1 MOSIG

UE Parallel Algorithms and Programming

Lab # 5

F. Desprez (Inria), J-F. Méhaut (UGA/Polytech), T. Ropars (UGA/IM^{2}AG), E. Saillard (Inria)

April 4th, 2017

1 Important information

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: **April 10th, 2017**.
- The assignment is to be turned in on Moodle (M1 Mosig Course: Parallel Algorithms and Programming, pwd: PAP_2017)
- The number of points associated with each exercise is only here to give you an idea of the weight associated with each exercise (The maximum grade is 20, there are extra points).

2 Your submission

Your submission is to be turned in on Moodle as an archive named with the last name of the two students involved in the project: `Name1_Name2_lab5.tar.gz`.

The archive will include:

- A report (in `pdf` format[1]) that should include the following sections:
  - The name of the participants.
  - For each exercise:
    - A short description of your work (what you did and didn't do)
    - Any additional information that you think is required to understand your code.
    - Your answer to the questions listed for each exercise.
- Your properly commented source code for each exercise

[1] Other formats will be rejected

Introduction

Let us consider $p = q^2$, p being the number of processors. Processors are organized in a 2D grid of size $q \times q$ (a 2D torus, to be more precise).

In this lab, we are going to work on the matrix multiplication problem $C = A \times B$. We denote by $[A, B, C]_{ij}$ the blockks of matrices A,B,C assigned to process P_{ij}, i and j being the coordinates of process P_{ij} in the grid.

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

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

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

Figure 1: Matrices distribution

Exercise 1: Fox matrix product algorithm (10 points)

Write a MPI program that implements a matrix product using Fox algorithm. Fox algorithm is described in Algorithm 1.

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 2 and 3.

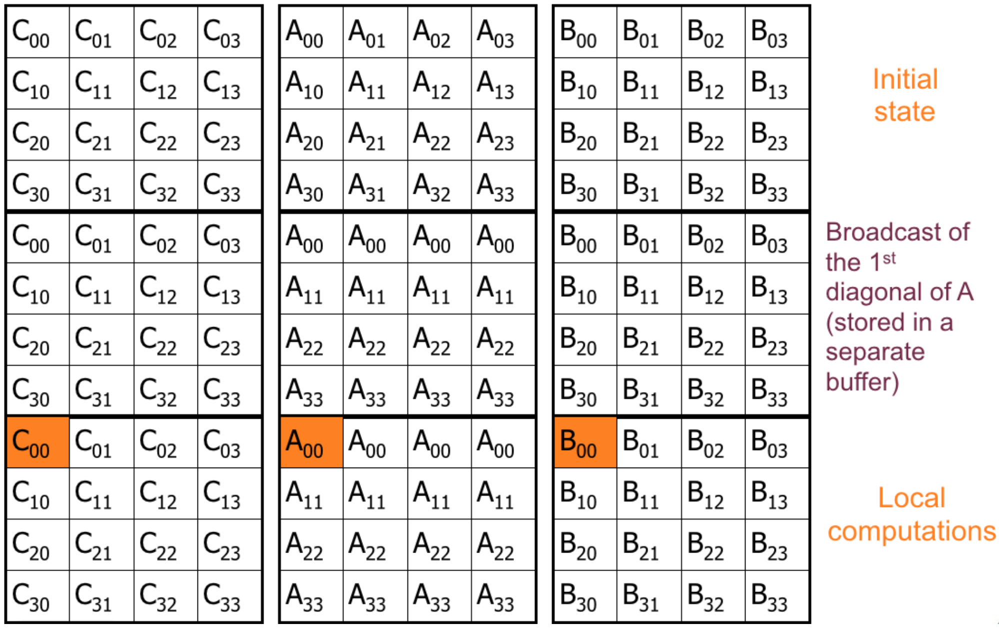

Figure 2: Step 1 of Fox algorithm

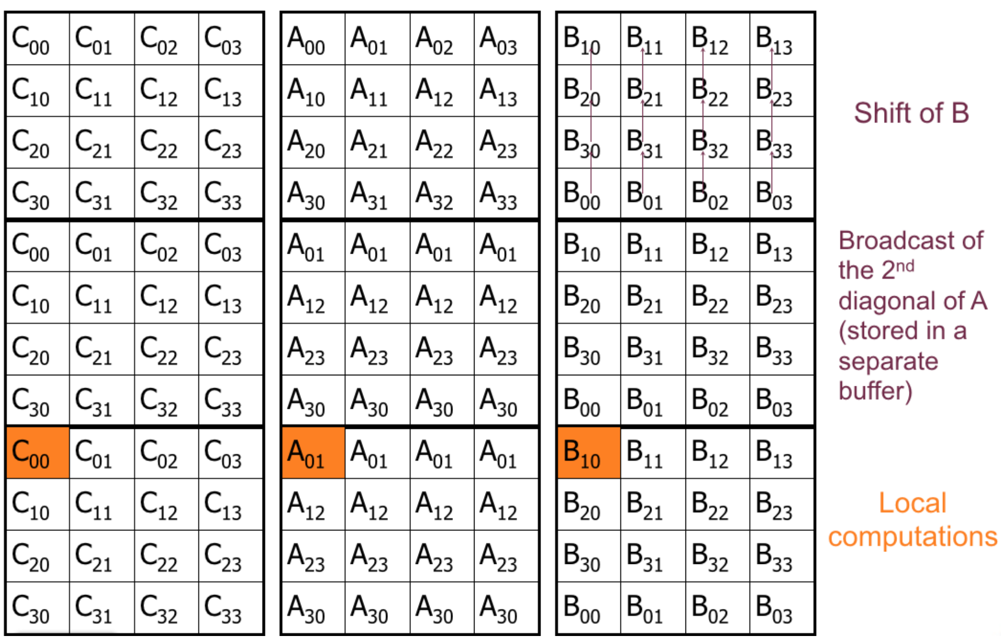

Figure 3: Step 2 of Fox algorithm

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.

Answer to the following points in your report:

1. Describe the optimizations that you included in your code if any.
2. Suggest other optimizations that you would have liked to implement.
3. Discuss the complexity of the algorithm you implemented

Exercise 2: Cannon matrix product algorithm (6 points)

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 4). Then matrix multiplication is run as described in Figure 5. 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 4: After the preskewing of A and B

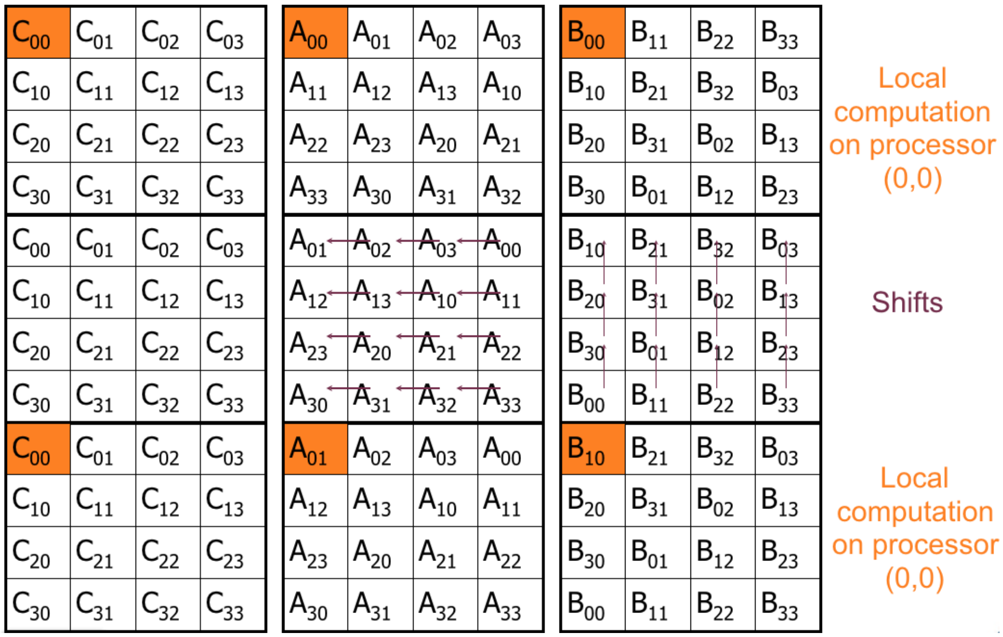

Figure 5: Steps of the Canon algorithm

About data management (IMPORTANT): In this exercise, we make the same assumptions regarding data management as in Exercise 1:

- 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.

Answer to the following points in your report:

1. Describe the optimizations that you included in your code if any.
2. Suggest other optimizations that you would have liked to implement.
3. Discuss the complexity of the algorithm you implemented

Exercise 3: Data management (6 points)

In this exercise, you are asked to implement a new version of the two algorithms, 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()`

Exercise 4: Performance evaluation (6 points)

You will find on Moodle a description of the *Froggy* cluster, as well as a short guide that explains how to use it.

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

A few suggestions about the experiments on Froggy:

- Considering the small number of resources available to you on Froggy, only consider small distributed setups for evaluation. For instance, 4 processes using 2 cores on 2 different nodes, and 9 processes using 3 cores on 3 different machines.
- Select appropriate matrix size to avoid too short and too long runs.

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`