HUST-USYD Summer School on Parallel Programming Practice – Lecture 8

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

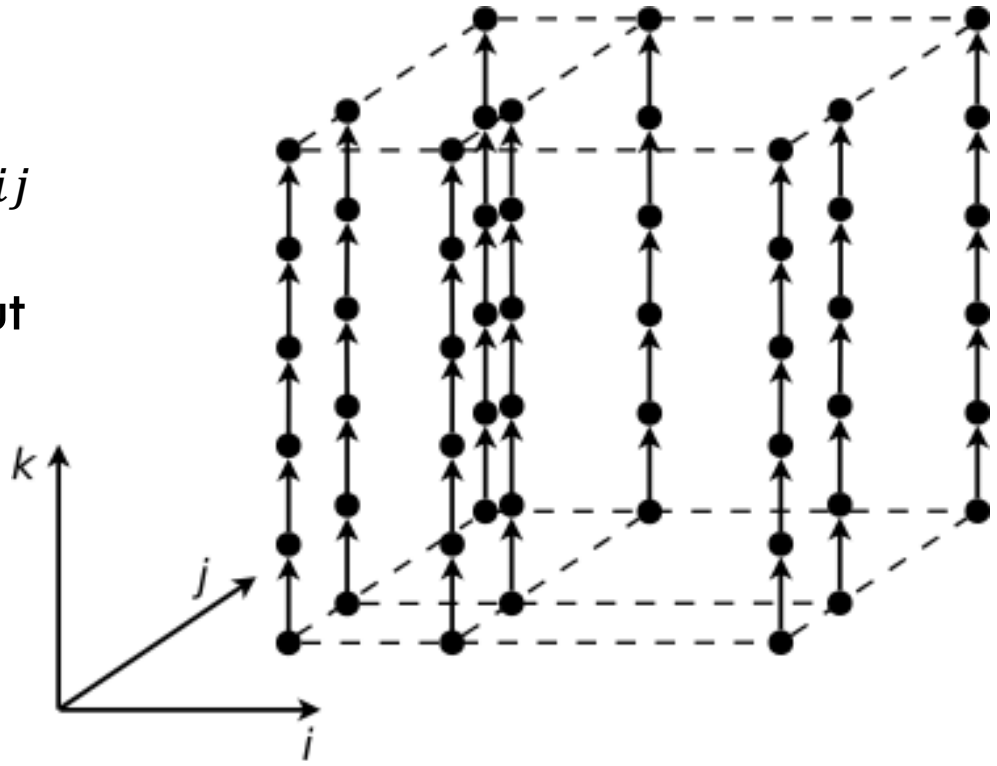
THE UNIVERSITY OF SYDNEY

Outline

- Review of Homework 5
- Parallel algorithms for matrix multiplication on distributed-memory 1D array /ring
- MPI: Message Passing Interface
  - The minimal set of MPI routines
- Lab exercises
- Homework 6

Matrix Multiplication

- Task dependency graph:
  - Data dependency for calculating each individual c_{ij}
  - All output elements c_{ij} can be computed simultaneously without any dependency

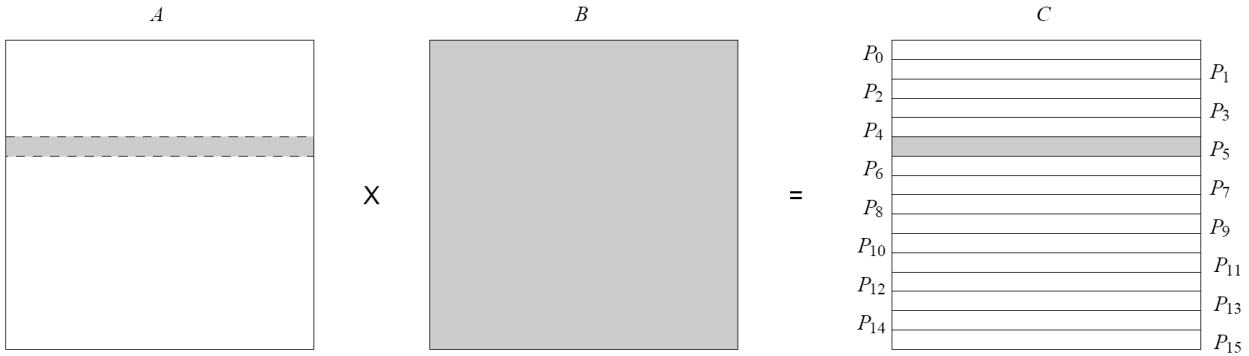


Partitioning

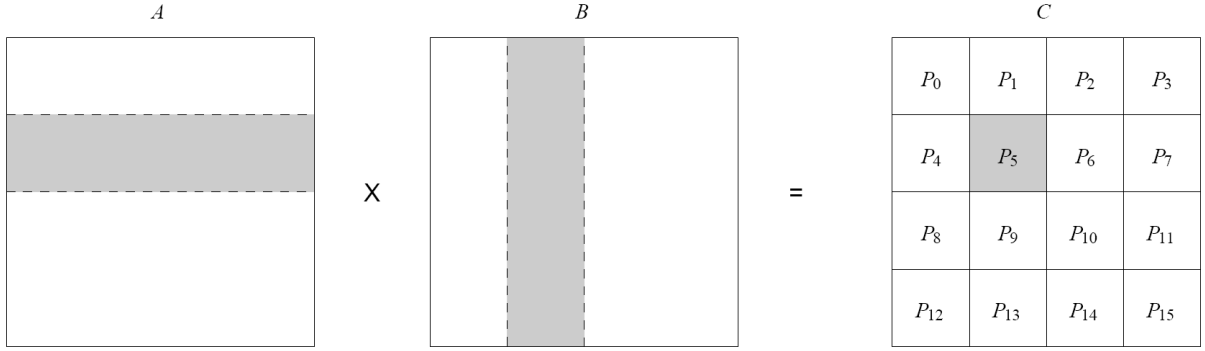
- In the algorithm design for shared memory platform, we discussed how to partition task matrix C and then associate input data with computation
  - Input matrices are shared, don't need to be moved around in main memory
- The partitioning techniques, 1D blocking, or 2D blocking can be used
- For distributed memory machines, however, input data matrices A and B also need to be partitioned and distributed to different processes
- Key issues: how to partition both tasks and data, and how to assign them to processes, and more importantly
- How to move data among processes to ensure
  - right data arrive in right processes at right time
  - communication overhead also minimized

Task and Data Association

- In 1D blocking a block row of C (task) is associated with a block row of A and the whole B:

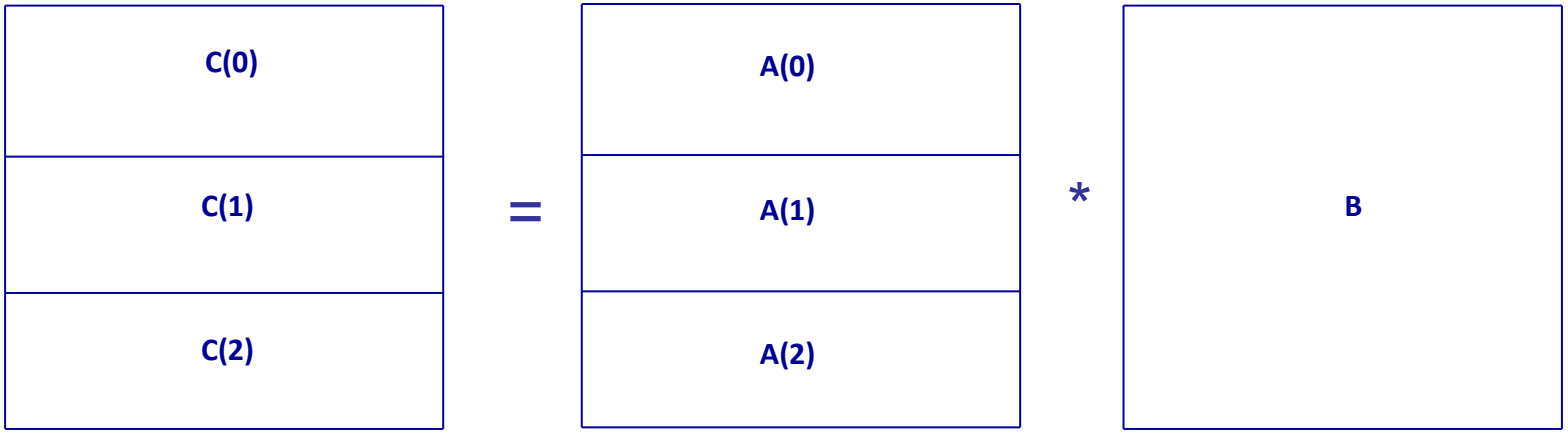


- In 2D blocking a block of C (task) is associated with a block row of A and a block column of B



Parallel MM on 1D Array/Ring

- First let's consider how to design parallel algorithms on a 1D array/ring
  - Processing elements are organized as a linear array/ring
- Review block matrix multiplication:
  - Partition matrices C and A into p x 1 block matrices, each block consists of n/p rows
  - Let $C(i)$ and $A(i)$ represent each block (call it block row) of size n/p x n
  - We have $C(i) = C(i) + A(i) * B$



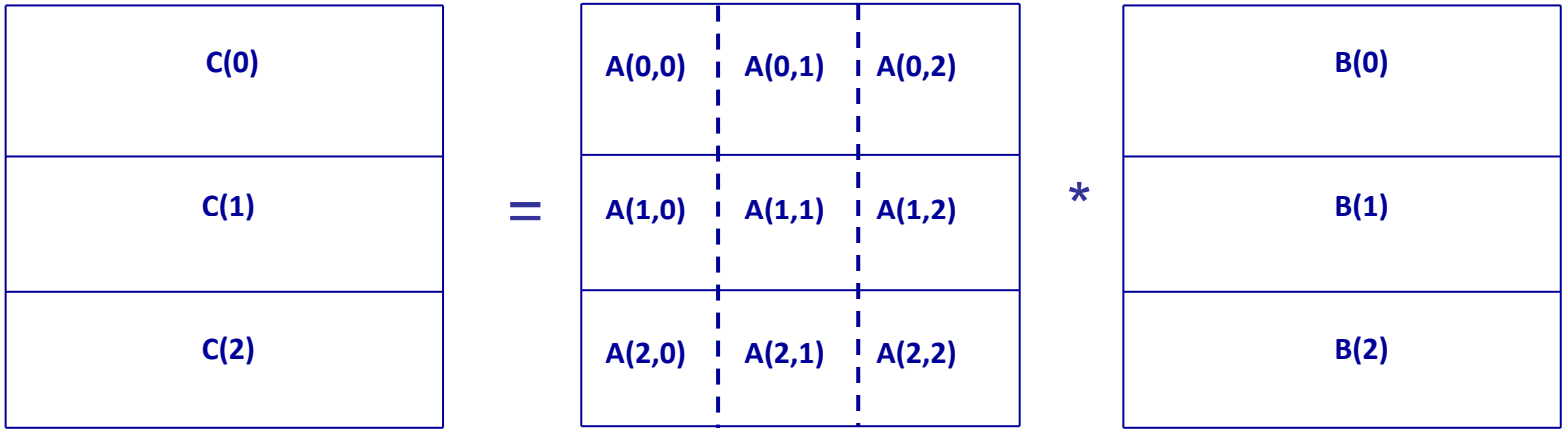
Parallel MM on 1D Array/Ring

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

- Further partition $A(i)$:
  - $A(i,j)$ is the n/p by n/p sub-block of $A(i)$
  - in columns $j * n/p$ through $(j+1) * n/p - 1$
- Also partition B into p x 1block matrix
  - #columes in $A(i,j)$ must equal #rows in $B(i)$
- Then we have

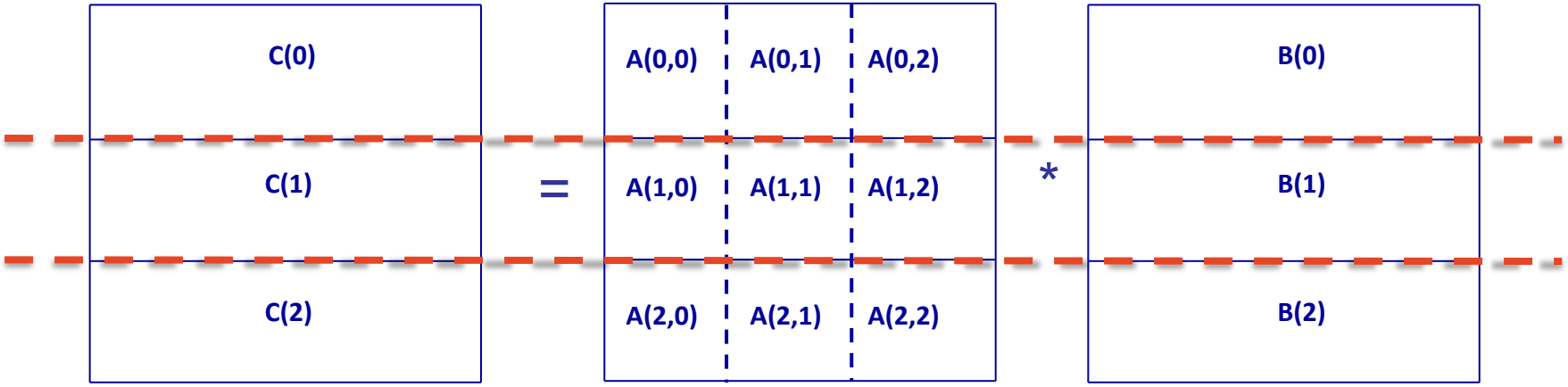
$$C(i) = C(i) + A(i) * B = C(i) + \sum_j \mathrm{A(i,j)} * \mathrm{B(j)}$$

  - e.g., $C(0) = C(0) + A(0,0) * B(0) + A(01) * B(1) + A(02) * B(2)$

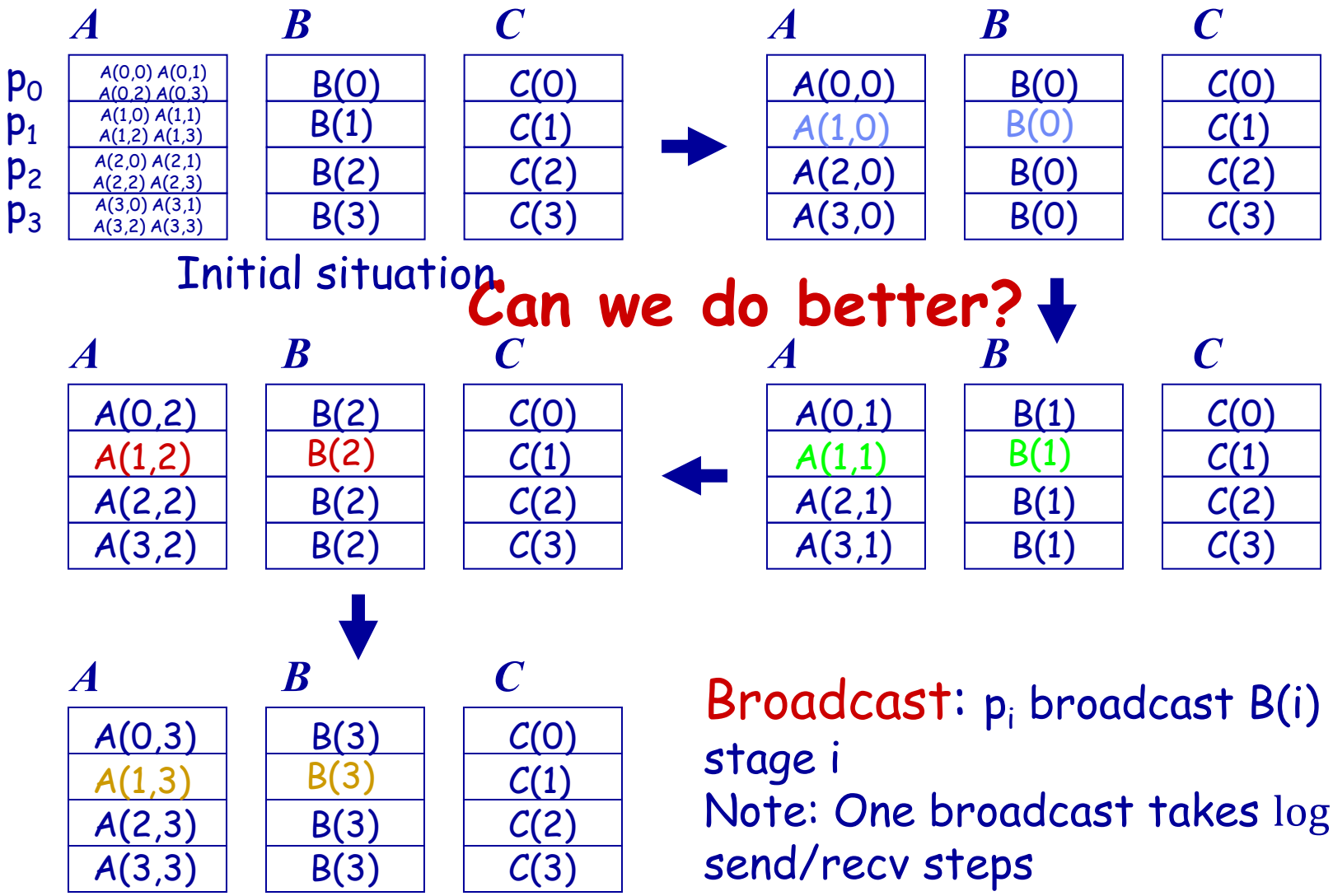


Parallel MM on 1D Array/Ring

- Assume that we have p processors which are organized as a 1D array/ring
- We partition each matrix (A, B, and C) as row block matrix and assign each block row ($A(i)$, $B(i)$ and $C(i)$) to one processor
- To compute each $C(i)$ we need $A(i)$ and the whole B
  - i.e., $C(i) = C(i) + A(i) * B = C(i) + \sum_j \mathrm{A(i,j)} * \mathrm{B(j)}$
- However, $B(i)$s are distributed to different processors
- we need explicit communication – how?



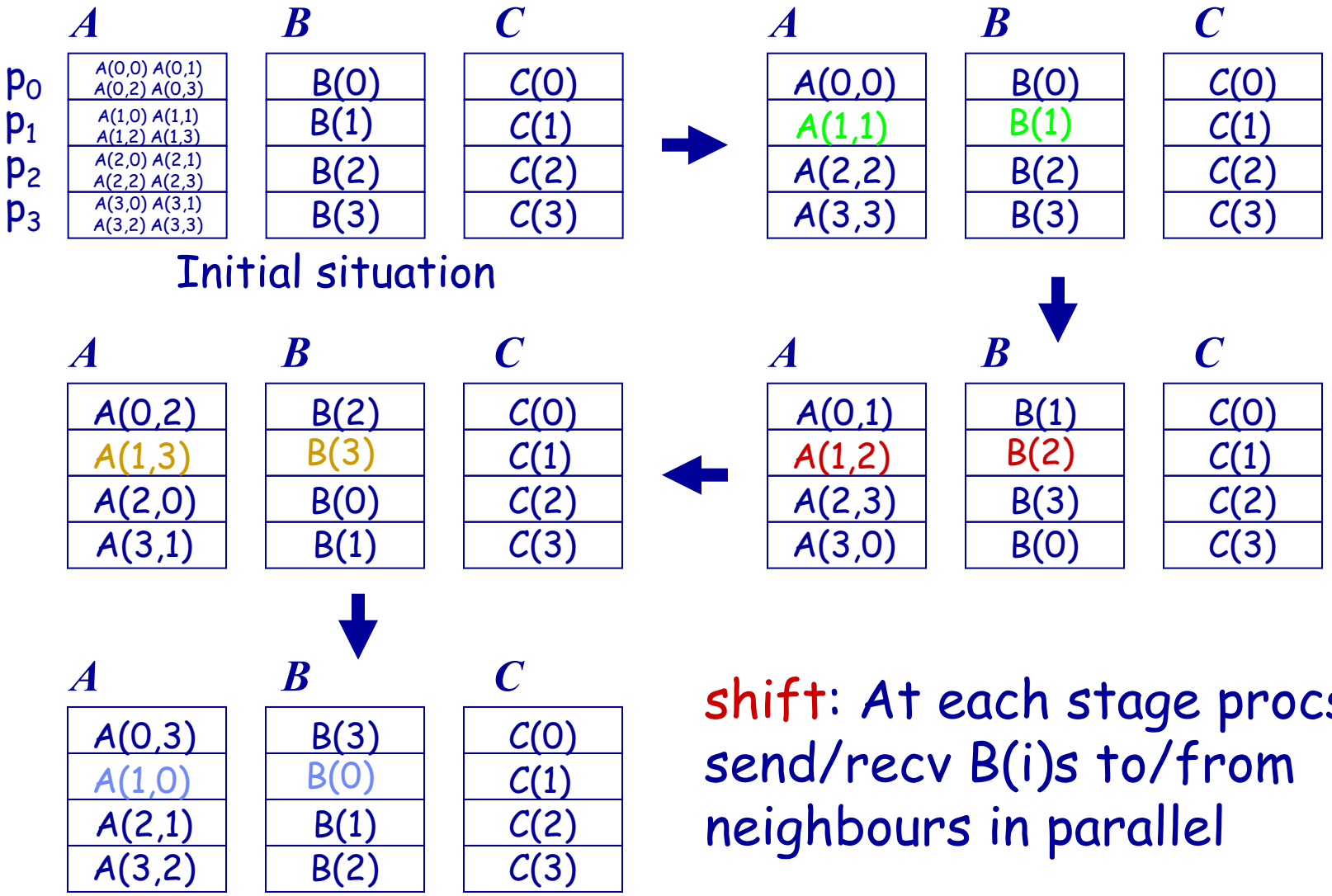
Parallel MM on 1D Array/Ring



Broadcast: p_i broadcast B(i) at stage i

Note: One broadcast takes $\log p$ send/recv steps

Parallel MM on 1D Array/Ring



shift: At each stage procs send/recv B(i)s to/from neighbours in parallel

MPI: Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines
- Vendor implementations of MPI are available on almost all commercial parallel computers
- The Standard itself:
  - at http://www.mpi-forum.org
- Other information on Web:
  - http://www.mcs.anl.gov/research/projects/mpi/index.htm
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

MPI: Message Passing Interface

- Subroutines for
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives: all processors get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, prefix sum, … of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to processes
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?

SPMD: Single Program Multiple Data

- All processes run the same program on different parts and data on P processing elements where P can be arbitrarily large
  - MPI programs use this pattern
  - Use the rank, or ID ranging from 0 to $(P-1)$ … to select between a set of tasks and to manage any shared data structures
- It is probably the most commonly used pattern in the history of parallel programming

The Minimal Set of MPI Routines

- Most MPI programs use the following six routines:
  - MPI_Init — Initializes MPI
  - MPI_Finalize — Terminates MPI
  - MPI_Comm_size — Determines the number of processes
  - MPI_Comm_rank — Determines the label of calling process
  - MPI_Send — Sends a message
  - MPI_Recv — Receives a message.

Starting & Terminating MPI Library

- The prototypes of the starting/terminating functions are:
- int MPI_Init(int *argc, char **argv)
- int MPI_Finalize()
- MPI_Init is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment
- MPI_Finalize is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment
- MPI_Init also strips off any MPI related command-line arguments
- All MPI routines, data-types, and constants are prefixed by "MPI_". The return code for successful completion is MPI_SUCCESS

Communicators

- A communicator defines a **communication domain** - a set of processes that are allowed to communicate with each other
- Information about communication domains is stored in variables of type MPI_Comm
- Communicators are used as arguments to all message transfer MPI routines
- A process can belong to many different (possibly overlapping) communication domains
- MPI defines a default communicator called MPI_COMM_WORLD which includes all the processes

Querying Information

- int MPI_Comm_size(MPI_Comm comm, int *size)
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
- The MPI_Comm_size and MPI_Comm_rank functions are used to determine the number of processes and the label of the calling process, respectively
- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one
  - Process's rank (or id) is essential in the SPMD model

Our First MPI Program

```
#include <mpi.h>
…
main(int argc, char *argv[]){
   int npes, myrank name_len;
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Get_processor_name(processor_name, &name_len);
   printf("Hello world from %s, rank %d/%d\n", processor_name, myrank, npes);
   MPI_Finalize();
}
```

- To compile the program using mpicc:

  mpicc –o myprog myprog.c
- To run the program using mpirun:
- mpirun –np 5 myprog

Sending & Receiving Messages

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

- The MPI_Send and MPI_Recv are the basic functions for sending and receiving messages in MPI
- Data are presented by triplet {buf, count, datatype}
  - MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons
- The dest/source is receive/send process's rank in a communicator comm (default is MPI_COMM_WORLD)

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Sending & Receiving Messages

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- The message-tag can take values ranging from zero up to the MPI defined constant MPI_TAG_UB
- MPI allows specification of wildcard arguments for both source and tag
  - If source is set to MPI_ANY_SOURCE, then any process of the communication domain can be the source of the message
  - If tag is set to MPI_ANY_TAG, then messages with any tag are accepted

Sending & Receiving Messages

- On the receive side, the message must be of length equal to or less than the length field specified
- The process is blocked in the MPI function until:
  - For receives the remote data has been safely copied into the receive buffer
  - For sends the send buffer can be safely modified by the user without impacting the message transfer

Deadlocks

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Recv(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD, &status);
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    MPI_Send(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
}
...
```

- MPI_Recv is blocking, there is a deadlock

Deadlocks

- In the following code, process i receives a message from process i - 1 (module the number of processes) and sends a message to process i + 1 (modulo the number of processes)

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
        MPI_COMM_WORLD, &status);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes,
        MPI_COMM_WORLD);
...
```

- Again, we have a deadlock!

Avoiding Deadlocks

– We can break the circular wait to avoid deadlocks:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD, &status);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD, &status);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Sending and Receiving Simultaneously

- To exchange messages, MPI provides the following function:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

- The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag, int source,
        int recvtag, MPI_Comm comm,  MPI_Status *status)
```

Lab Exercise 1: send/recv Communication Program 1

- Compile and run "mpi_comm0.c" program
- Modify the program as described in the following:
- Processes are organized as a 1D ring
- They shift their IDs around, e.g., 0 -> 4 -> 3 -> 2 -> 1 -> 0
  - i.e., initially each process sends its own ID to its left neighboring process
  - Once receiving an integer from its right neighbor, the process will send the same integer received to its left
- In numprocs-1 rounds every process will have received all IDs of other processes once
- Only process 0 collects (stores in an array of size numprocs) the received IDs and finally prints them out

Sending & Receiving Messages

- On the receiving end, the status variable can be used to get information about the MPI_Recv operation
- The corresponding data structure contains:

```
typedef struct MPI_Status {
int MPI_SOURCE;
int MPI_TAG;
int MPI_ERROR; };
```

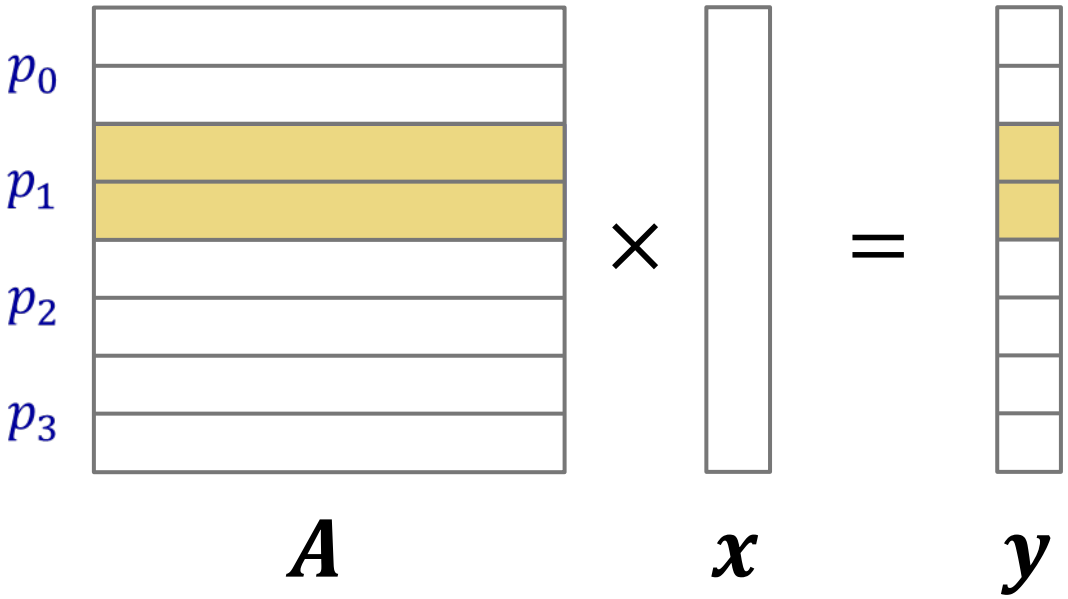
- The MPI_Get_count function returns the precise count of data items received
  - int MPI_Get_count(MPI_Status *status,
    MPI_Datatype datatype, int *count)

Lab Exercise 2: send/recv Communication Program 2

- Modify the program you have done for Lab Exercise 1 as follows:
- Processes are still organized as a 1D ring
- This time each process has a number of integers
  - I.e., process i has i+1 integers and all integers are set to the same value as its rank
  - E.g., process 4 has 5 integers and all 5 integers have the same value equal to 4
- They shift their integers around, e.g., 0 -> 4 -> 3 -> 2 -> 1 -> 0 and again in numprocs-1 rounds every process will have received all integers of other processes once
- Only process 0 stores all the received in an arrang and at the end prints them out
- Note: each time different processes sent different number of integers

Matrix-Vector Multiplication

- One parallel algorithm: (many other options)
  - Partition matrix A into p blocks of rows
    - block matrix (n/p by m)
  - Partition vector y into p blocks – block vector (n/p by 1)
  - One block A and associated one block y to a processor
  - Not partition vector x, but broadcast x to every processor



Lab Exercise 2: send/recv Communication Program 3

- Write an MPI program to compute matrix-vector multiplication y = Ax
- Assume matrix A is of size n by m
- Your program needs to take n, and m as command-line arguments
- In your program
  - Process 0 generates matrix A and x, partition the matrix into numprocs row blocks of equal size (the difference must not be greater than 1for load balancing) and then sends one block to a process
  - Process 0 also sends vector x to all other processes
  - After receiving the block and vector from Process 0, each process will compute partial results and then send results to Process 0
  - Process 0 collect the partial results from other processes and check the correctness

Homework 6

- Write an MPI program to compute matrix multiplication

$$C = AB$$

- Assume matrix A is of size n by l and B is of size l by m
- Your program needs to take n, l, and m as command-line arguments
- In your program
  - Assume that the processors are organized in a 1D ring/array
  - Process 0 generates input matrices A and B, partitions the matrices into numprocs row blocks of equal size (the difference must not be greater than 1for load balancing) and then sends each process one block of A and one block of B
  - After the computation other processes will send the results to Process 0
  - Process 0 collects the partial results from other processes and checks the correctness

THE UNIVERSITY OF SYDNEY