HUST-USYD Summer School on Parallel Programming Practice – Lecture 9

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

School of Computer Science, University of Sydney

Outline

- Non-blocking send/recv
- Collective communication routines
- Lab exercise: Matrix-vector multiplication revisited
- Parallel matrix multiplication on 2D mesh/torus
- MPI Cartesian topology routines
- Homework 7

Non-Blocking Send & Receive

- MPI Functions: MPI_Isend, MPI_Irecv
- int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
- int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- When MPI_Isend/MPI_Irecv is called, the calling process will be not blocked and it continues to execute the subsequent instructions

- MPI_Request *request: a so called opaque object, which identifies communication operations and matches the operation that initiates the communication with the operation that terminates it

Non-Blocking Send & Receive

- MPI Functions: MPI_Test, MPI_Wait
- int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
- int MPI_Wait(MPI_Request *request, MPI_Status *status)
- MPI_Test tests if operation finished
  - It not, the calling process will continue its execution
- MPI_Wait blocks the calling process until the non-blocking send/recv operation is finished

Non-Blocking Send & Receive

- MPI Functions: MPI_Testsome, MPI_Waitsome
- int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)

- int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)

- Tests or waits until at least one of the operations associated with active handles in the list have completed
- Needs an array of requests (also indices and statuses) for multiple Isend/Irecv operations

Collective Communication

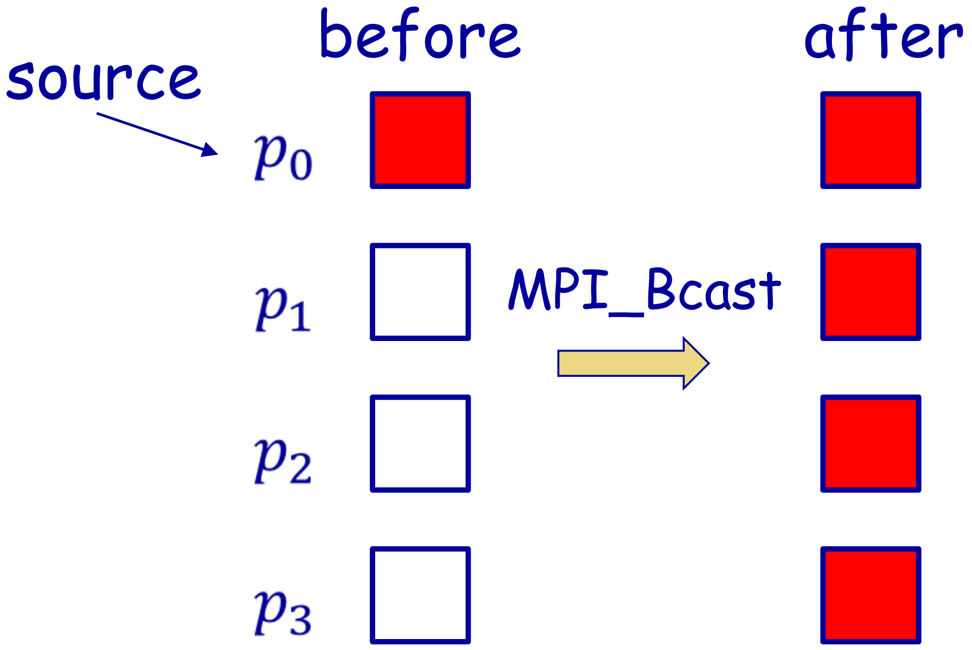
- MPI specifies a set of collective communication functions
  - One-to-All: One process contributes to the result and all processes receive the result
    - MPI_Bcast, MPI_Scatter, MPI_Scatterv, …
  - All-to-One: All processes contribute to the result and one process receives the result
    - MPI_Gather, MPI_Gatherv, …
  - All-to-All: All processes contribute to the result and all processes receive the result
    - MPI_Allgather, MPI_Alltoall, MPI_Allgatherv, …
  - Other: MPI_Barrier, MPI_Reduce, MPI_Scan, …

Broadcast

- The MPI broadcast routine (one-to-all) is:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int source, MPI_Comm comm)
```

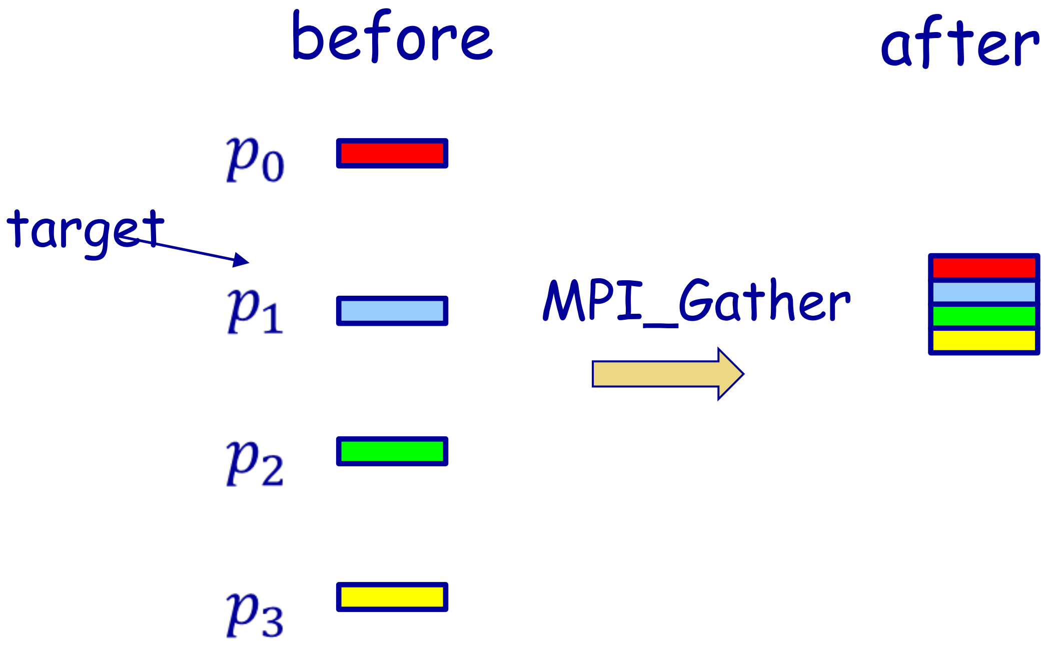
- All processes must call MPI_Bcast()



Gather

– The MPI gather operation (all-to-one) is:

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
```



Gather

- The MPI gather operation (all-to-one) is:

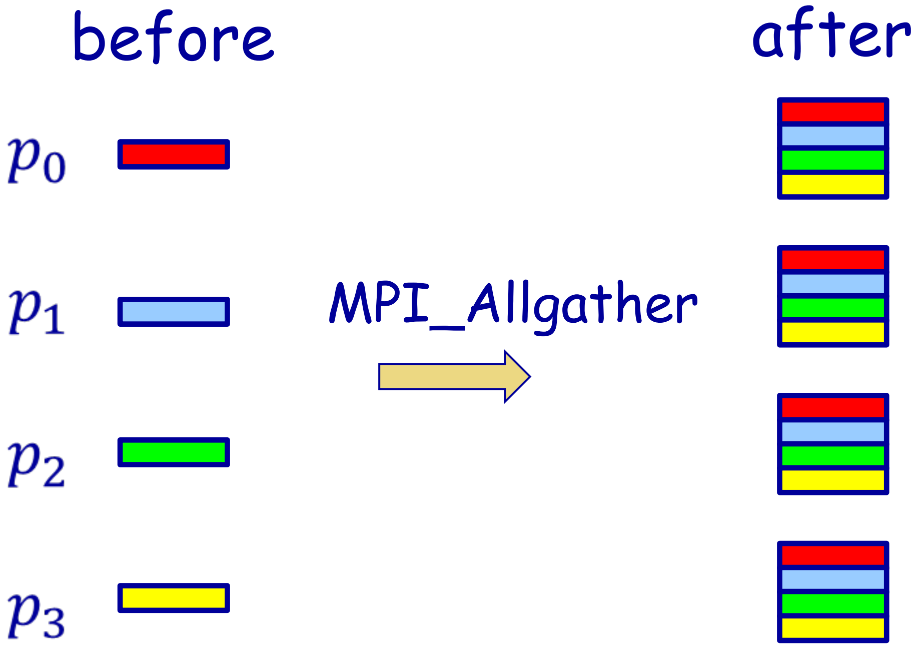
```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
```

- sendcount – number of elements each process sends
- recvcount – number of elements the target process will receive from each process
- *recvbuf – the length of the recvbuf must be at least recvcount times the number of processes involved in the operation

Allgather

– The MPI All gather operation (all-to-all) is:

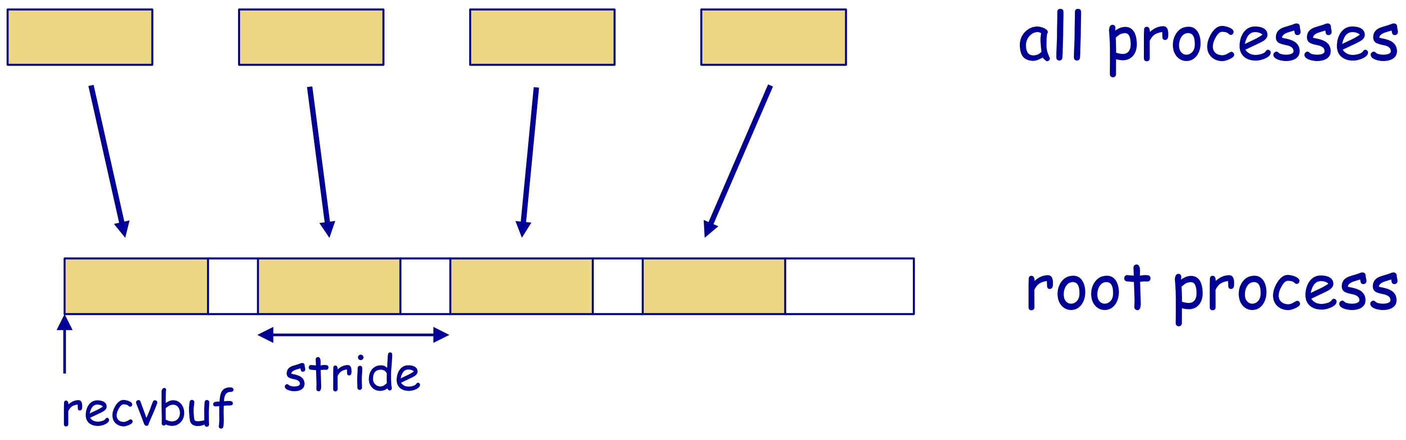
```
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```



Gatherv

– The MPI gatherv operation (all-to-one) is:

```
int MPI_Gatherv(void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                int *recvcounts, int *displs, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```



Gatherv

– The MPI gatherv operation (all-to-one) is:

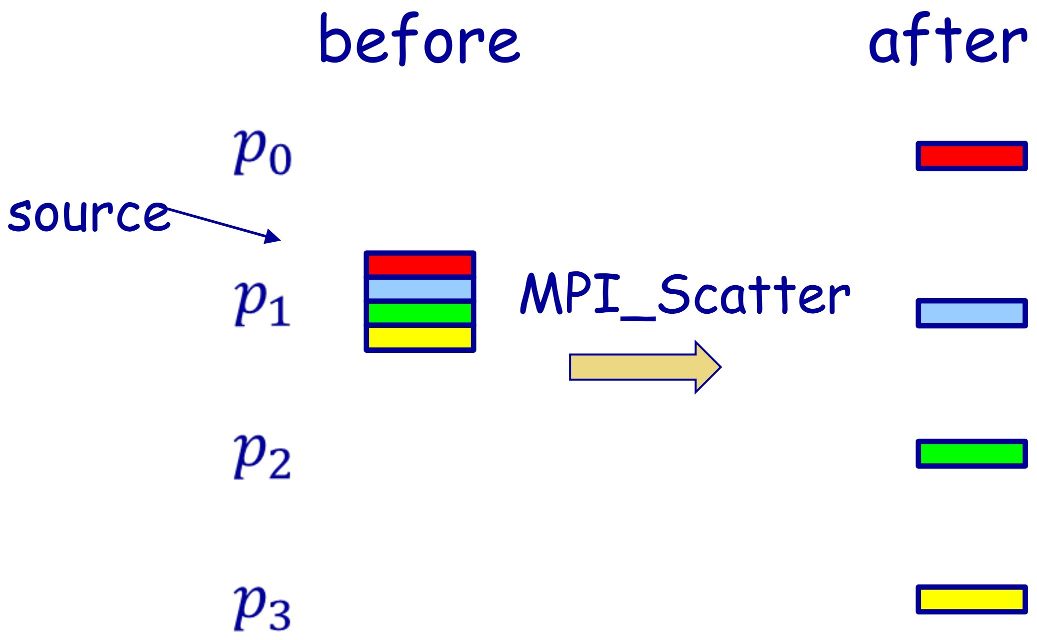
```
int MPI_Gatherv(void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                int *recvcounts, int *displs, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

```
int sendarray[100];
int stride = 150;
int *displs, *rcounts;
…
rbuf = (int *) malloc(numprocs*stride*sizeof(int));
for (int i=0; i<numprocs; i++){
    displs[i] = i * stride;
    rcounts[i] = 100;
}
MPI_Gatherv(senarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT, root,
            MPI_COMM_WORLD);
```

Scatter

– The MPI scatter operation (one-to-all) is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype senddatatype, void *recvbuf,
                int recvcount, MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```



Scatter

- The MPI scatter operation (one-to-all) is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype senddatatype, void *recvbuf,
                int recvcount, MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```

- sendcount – number of elements the source process will send to each process
- *sendbuf – the length of the sendbuf must be at least sendcount times the number of processes involved in the operation
- recvcount – number of elements each process will receive

Scatterv and Alltoall

– The MPI scatterv operation (one-to-all) is:

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root,
                 MPI_Comm comm)
```

– The MPI all-to-all communication operation is:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, void *recvbuf,
                 int recvcount,  MPI_Datatype recvdatatype,
                 MPI_Comm comm)
```

Other Collectives

– The barrier synchronization operation:

```
int MPI_Barrier(MPI_Comm comm)
```

– The MPI reduction operation (all-to-one) is:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int target,
               MPI_Comm comm)
```

– If the result of the reduction operation is needed by all processes, MPI provides:

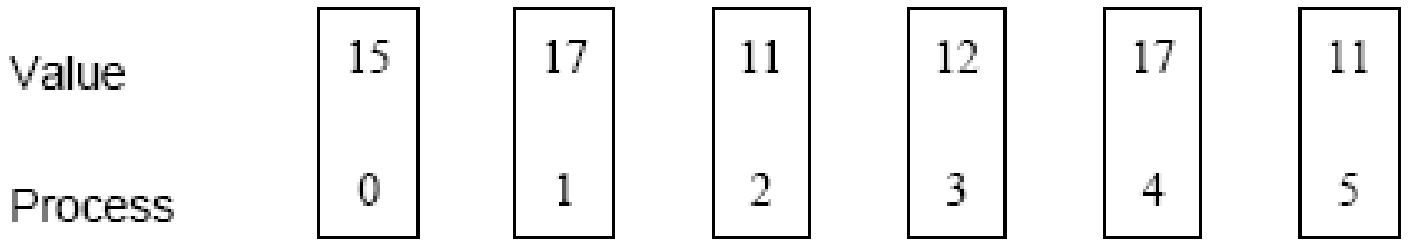
```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
     MPI_Datatype datatype,  MPI_Op op, MPI_Comm comm)
```

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	Max, value-location	Data-pairs
MPI_MINLOC	Min, value-location	Data-pairs

Reduction Operations

- The operation MPI_MAXLOC combines pairs of values (vi, li) and returns the pair (v, l) such that v is the maximum among all vi's and l is the corresponding li (if there are more than one, it is the smallest among all these li 's)
- MPI_MINLOC does the same, except for minimum value of vi



```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

Reduction Operations

– MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations

MPI Datatype	C Datatype
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_INT	long and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int

Reduction Operations

– Example:

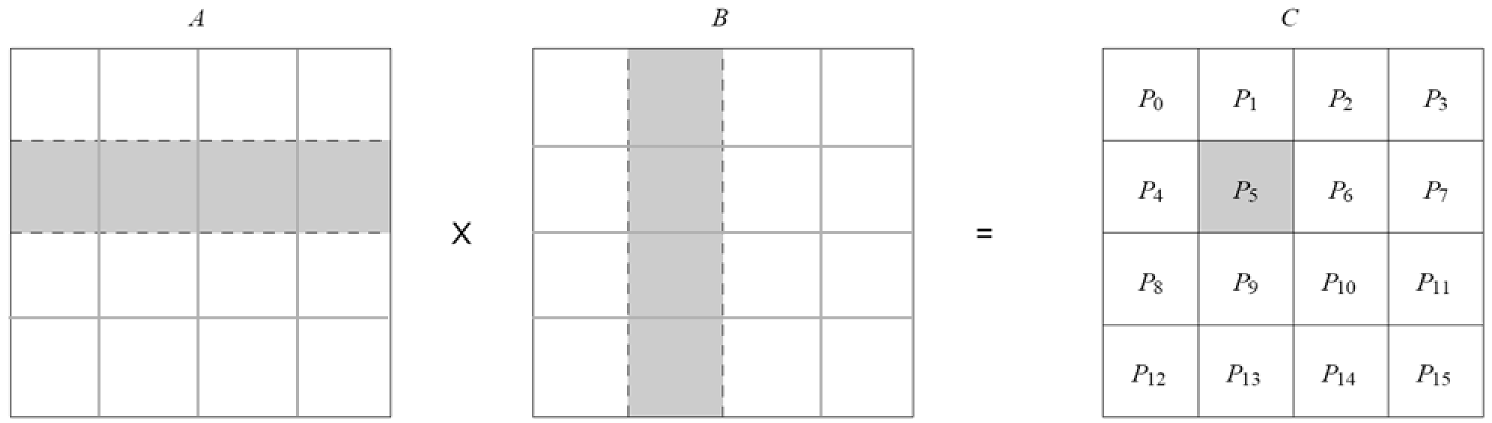
```
struct {
    double val;
    int   rank;
} in, out;

in.val = ain;
in.rank = myrank;
MPI_Reduce( in, out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root, 
comm );

if (myrank == root) {
    aout = out.val;
    ind = out.rank;
}
```

Lab Exercise: Matrix-Vector Multiplication Revisited

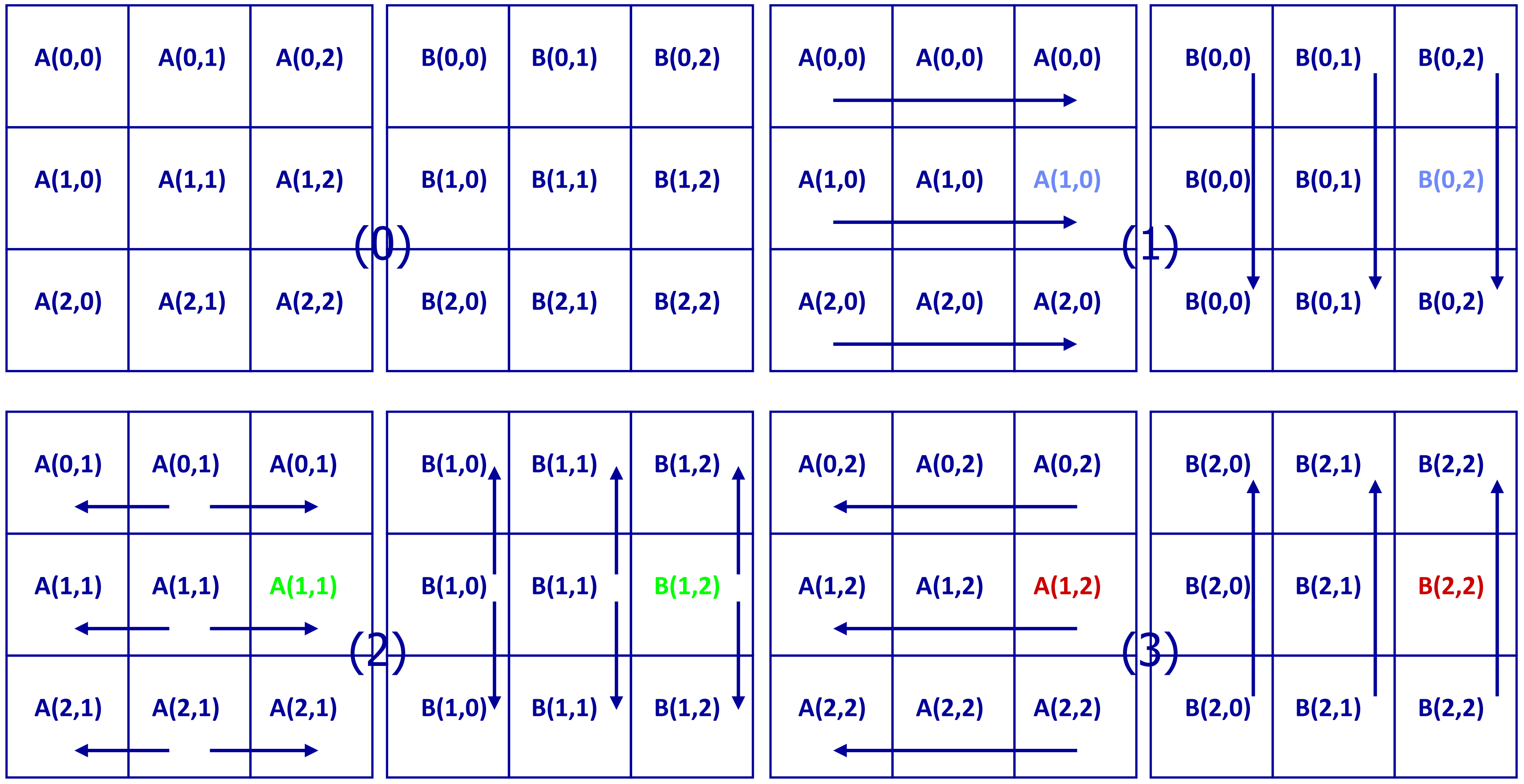
- Modify the program for matrix-vector multiplication you did in the last lecture by using collective communication routines
- You need to consider general cases, that is, n may not divisible by p
- Thus you need to use MPI_Scatterv and MPI_Getherv functions

Parallel MM on 2D Mesh/Torus

- Use 2D partitioning and assume $p = \sqrt{p} \times \sqrt{p}$, i.e., p processes are organized as a 2D mesh/torus
- Let $C(i,j)$ refer to a submatrix of size $n/\sqrt{p} \times n/\sqrt{p}$ and similar for $A(i,j)$ and $B(i,j)$
- Each $C(i,j)$ needs one row of $A(i,j)$s and one column of $B(i,j)$s which are held by different processes
- Communication: move $A(i,j)$s horizontally and $B(i,j)$s vertically

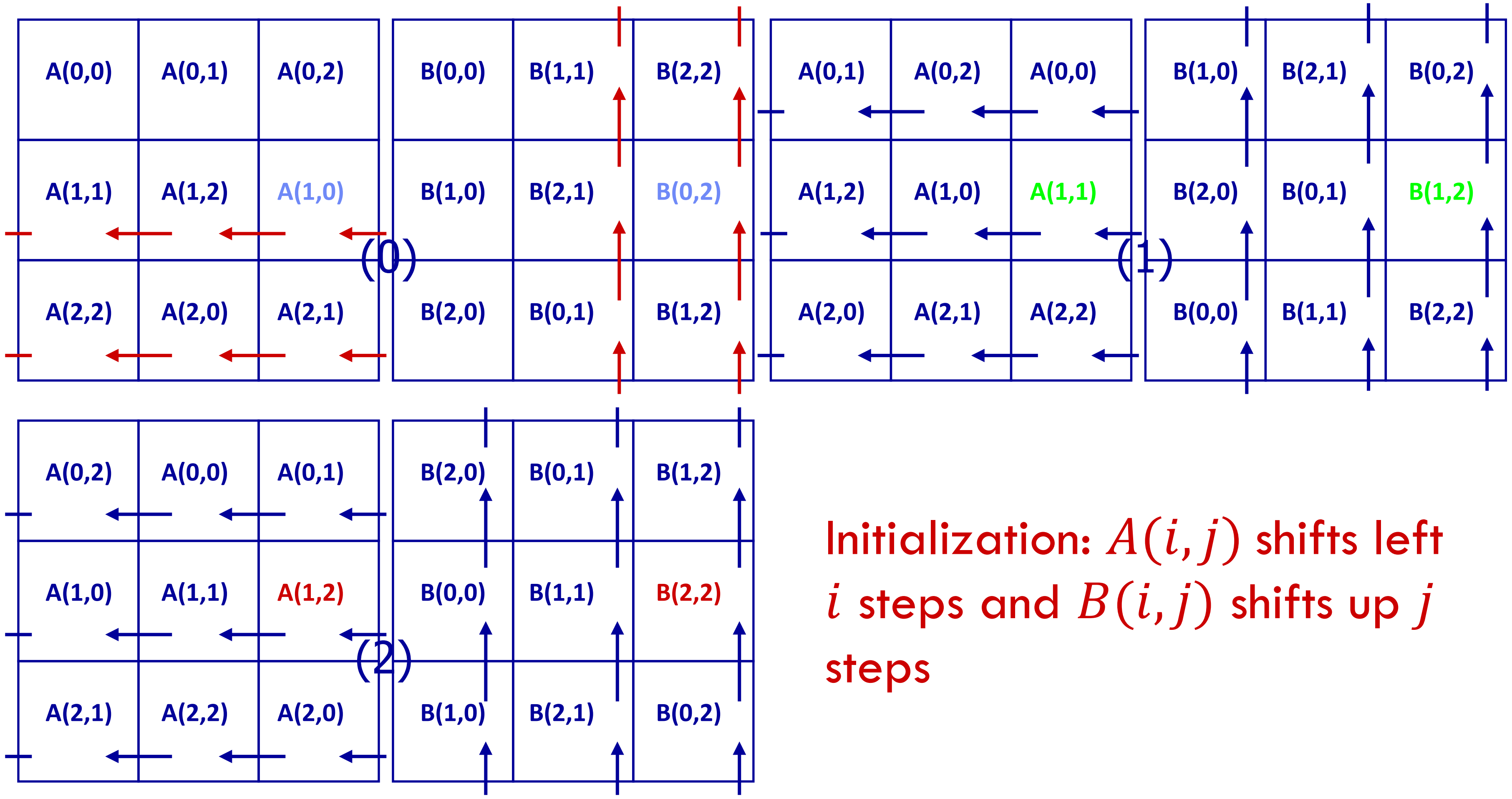


2D Algorithm 1 (Broadcast)



C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

2D Algorithm 1 (Cannon)



Initialization: $A(i,j)$ shifts left i steps and $B(i,j)$ shifts up j steps

C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

Parallel Matrix Multiplication

- We discussed 1D and 2D parallel algorithms for distributed-memory machines
- Question: which one performs better?
- 1D algorithm:
  - Matrices are partitioned into block rows $(n/p \times n)$
  - In one parallel step each process
    - Send one block row $(B(j))$ and received one block row $(\sim n^2/p)$
    - Multiplication of $A(i,j)$ $(n/p \times n/p)$ and $B(j)$ $(n/p \times n)$
      - Number of operations is $\sim n^3/p^2$
    - Computational intensity: $\sim n/p$
  - Number of parallel steps is p

Parallel Matrix Multiplication

- 2D algorithm:
  - Matrices are partitioned into a number of submatrices $(n/\sqrt{p} \times n/\sqrt{p})$
  - In one parallel step each process
    - Send one submatrix $A(i,j)$ and receive one submatrix horizontally $(\sim n^2/p)$
    - Send one submatrix $B(i,j)$ and receive one submatrix vertically $(\sim n^2/p)$
    - Multiplication of $A(i,k)$ and $B(k,j)$
      - Number of operations is $\sim n^3/(p\sqrt{p})$
    - Computational intensity: $\sim n/\sqrt{p}$
  - Number of parallel steps is $\sqrt{p}$

Parallel Matrix Multiplication

- Although 2D algorithm requires to move both $A(i,j)$ and $B(i,j)$ in a parallel step, the number of operations increased by a factor of $\sqrt{p}$
- The computational intensity is increased by a factor of $\sqrt{p}$
- Since the total amount of work is fixed for a given problem, the number of parallel steps is also reduced by a factor of $\sqrt{p}$
  - Greatly reduced the communication overheads
- Therefore, 2D parallel algorithm for matrix multiplication

MPI Cartesian Topology Routines

– MPI allows a programmer to organize processors into logical k-D meshes
– The processor ids in MPI_COMM_WORLD can be mapped to other communicators (corresponding to higher-dimensional meshes) in many ways
– The goodness of any such mapping is determined by the interaction pattern of the underlying program and the topology of the machine

Cartesian Topologies

- To create Cartesian topologies using the function:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                    int *periods, int reorder, MPI_Comm *comm_cart)
```

- This function takes the processes in the old communicator and creates a new communicator with ndims dimensions
- Each processor can now be identified in this new cartesian topology by a vector of dimension dims

Cartesian Topologies

– To determine process coords in cartesian topology given rank in group:

```
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
                   int *coords)
```

– To determine process rank in communicator by its cartesian location:

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

Cartesian Topologies

- To find the resulting source and destination ranks, given a shift direction and amount

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ,
                    int *source, int *dest)
```

- To partition a cartesian topology into subgrids

```
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[],
                    MPI_Comm* new_comm)
```

- MPI_Cart_sub is a collective comm routine
  - All involved processes must call

Homework 7: MPI Cartesian Topology

- Search the Internet for detailed descriptions on MPI Catesian topoloty
- Then write a simple MPI program to create 16 processes
- Use MPI cartesian routine to organize the processes into a 4 X 4 2D ring
- Declare two integers a and b
  - Assign process's id to a and then every process shifts a to the left neighbour process using MPI send/recv
  - Assign process's id to b and then every process shifts b up to the neighbor process above also using MPI send/recv
  - Print and check the results
- Partition processes into row and column subgrids
  - For processes with column id = 0, assign it original id to a and broadcast it to the processes in the same row
  - For processes with row id = 0, assign it original id to b and broadcast it to the processes in the same column
  - Print and check the results

Q&A

THE UNIVERSITY OF SYDNEY