COMP 4901Q: High Performance Computing (HPC)

Lecture 10: Collective Communication Algorithms in MPI

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)
Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)
Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

Outline

- Blocking vs. Nonblocking Communications
- MPI Derived Datatypes

- Collective Communications
  - Basic collective communications
  - Advanced collective algorithms

- A Case Study

Blocking Message Passing

- A blocking message passing operation means that it won't return until the desired operation has been finished.
  - It can guarantee the correctness of semantic.
  - Its performance may not be good, because while data is being transmitted, the sender/receiver can do nothing (i.e., being blocked!).

- MPI_Send and MPI_Recv are blocking operations.

- Case 1: blocking non-buffered Send/Receive
  - The send operation doesn't return until the matching receive has been encountered at the receiving process.

  - Once the receive operation is encountered, the message is sent and the send operation returns upon completion of the communication operation.

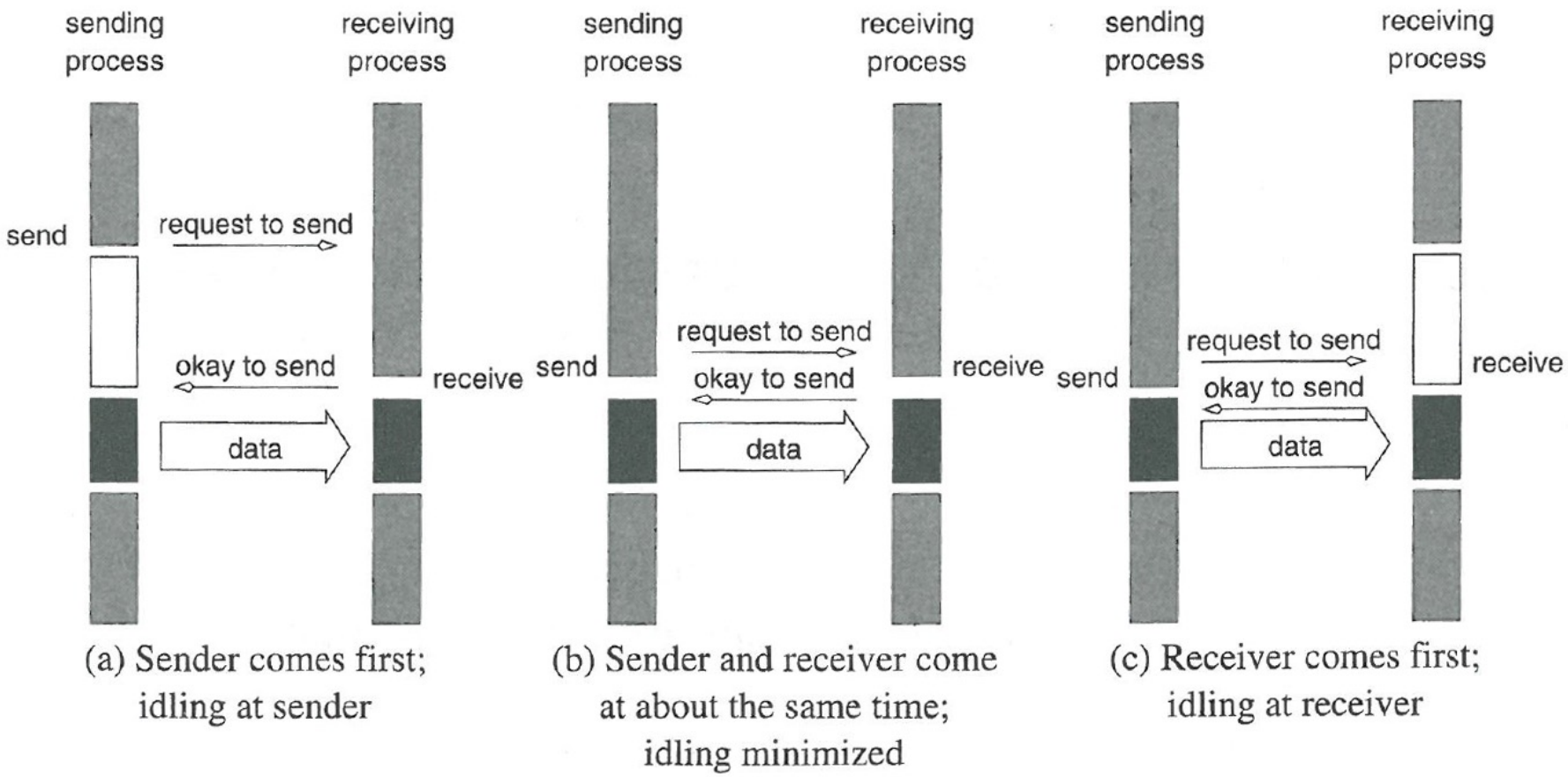
Blocking Non-Buffered Send/Receive

- Two issues of blocking non-buffered operations
  - Idling overhead
    - Since the send operation and receive operation may be executed at different time, either the sender or receiver will be idling for a period, waiting for the other side.
  - Deadlocks
    - The below example shows a deadlock scenario, in which two processes try to exchange two values, a and b.

Process 0	Process 1
Send(&a, 1, 1); Receive(&b, 1, 1);	Send(&a, 1, 0); Receive(&b, 1, 0);

Question: how to correct the above code?

Blocking Non-Buffered Send/Receive



(a) Sender comes first; idling at sender

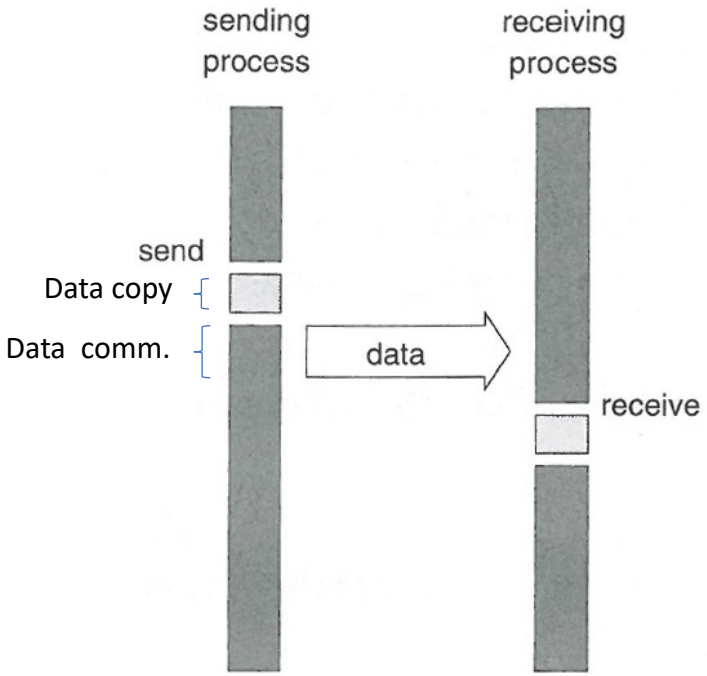
(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Source: Fig. 6.1 of Ref. [1].

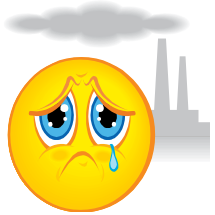
Blocking Message Passing

- Case 2: blocking buffered Send/Receive
  - The "idling overheads" and "deadlocks" issues of Case 1 can be alleviated by introducing "buffers".
- Example: The communication hardware has buffers at send and receive ends.
  - For the send operation, the sender copies the data into a buffer, and returns immediately after the copy operation.
  - The sender can continue with other jobs.
  - The actual communication can be accomplished in many ways, depending on the hardware resources.



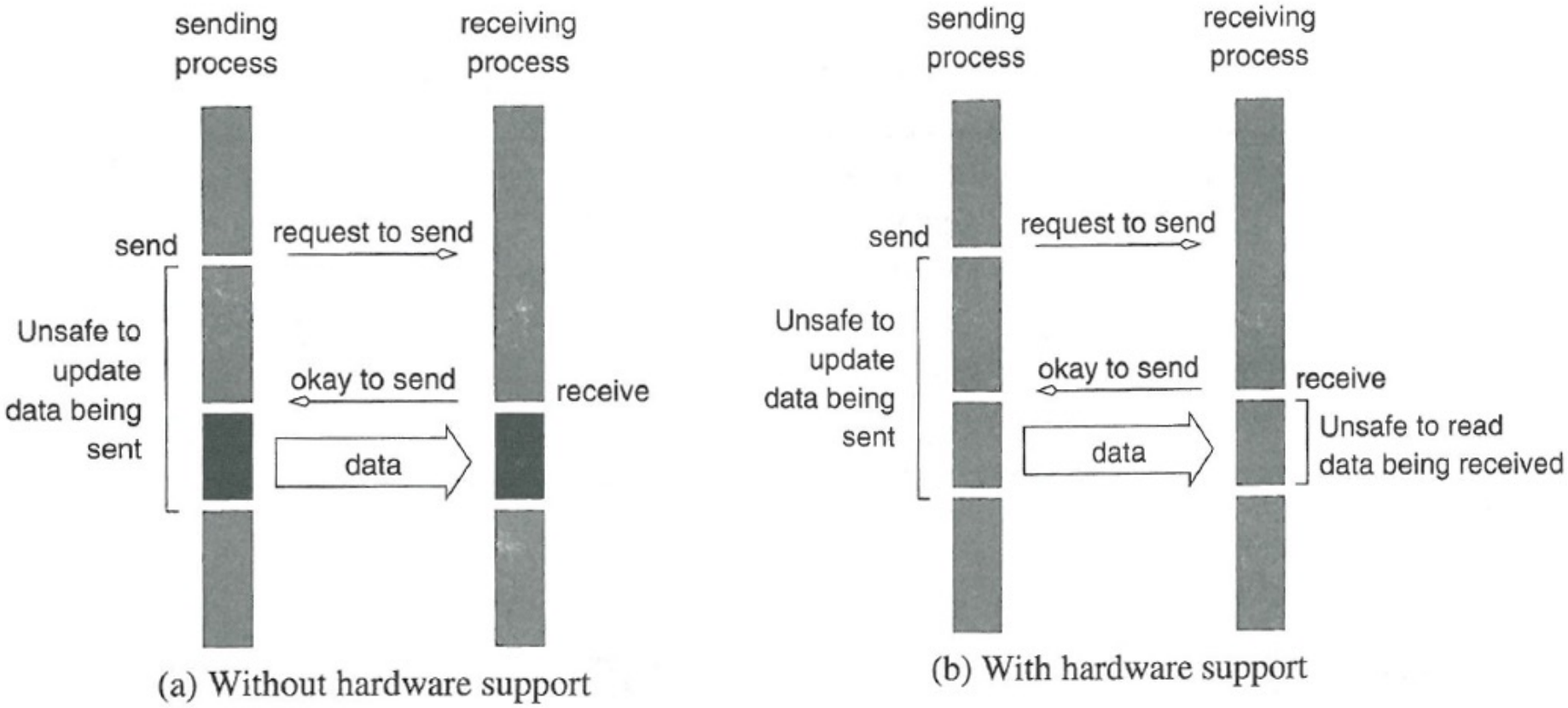
Source: Fig. 6.2 (a) of Ref. [1].

Non-blocking Message Passing

- Non-blocking send/receive operations return immediately, even if the communication is not finished.
  - It can avoid most deadlocks. 
  - It can achieve better performance by overlapping communication with computation. 
  - It requires a complicated program design. 

Non-blocking Message Passing

Non-blocking non-buffered send and receive operations



Source: Fig. 6.4 of Ref. [1].

Non-blocking Message Passing

- Problem: how does a process know that the data communication task is finished?

- Answer: the programmer needs to check the status of the previous non-blocking operation
  - If the data communication is finished, you can start to process the data;
  - If the data communication is not finished, you can either sleep or do some other safe computing tasks.

Non-blocking Message Passing in MPI

- MPI provides a pair of functions for non-blocking send and receive operations: MPI_Isend and MPI_Irecv

- int MPI_Isend(void* buf,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm,
  MPI_Request *request /* out */);
  - <buf, count, datatype> specify the message
  - dest specifies the rank of the process that should receive the message
  - tag (a nonnegative integer) is used to distinguish messages
  - comm specifies the process group
  - Request is a pointer to a request object, which is used by MPI_Test and MPI_Wait functions to identify the operation whose status we want to query or to wait for its completion.

Non-blocking Message Passing in MPI

- int MPI_Irecv(void* buf,
  int maxsize,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Request* request);
  - <buf, maxsize, datatype> specify where to store the message
  - source specifies the rank of the process from which the message should be received
  - tag should match the "tag" specified by the sender
  - comm specifies the process group
  - Request is a pointer to a request object, which is used by MPI_Test and MPI_Wait functions to identify the operation whose status we want to query or to wait for its completion.

- Remark: there is no status_p argument. The status information associated with the receive operation is returned by MPI_Test and MPI_Wait functions.

MPI_Test

- MPI_Test tests whether or not a non-blocking send or receive operation has finished.
  - The send or receive operation is identified by a request object with type MPI_Request

- int MPI_Test(MPI_Request *request,

  int *flag,

  MPI_Status *status);
  - **request** identifies the non-blocking operation. If the operation has finished, the request object will be deallocated.
  - **flag** is set to true (non-zero in C/C++) if the non-blocking operation has finished, or false (0 in C/C++) otherwise.
  - **status** is set to contain information about the operation if the non-blocking operation has finished.

MPI_Wait

- MPI_Wait blocks until the non-blocking operation (identified by an argument) completes.

- int MPI_Wait(MPI_Request *request,
  MPI_Status *status);
  - request identifies the non-blocking operation.
  - status is set to contain information about the operation.

A Simple Example

```
int main(int argc, char *argv[])
{
    int myid, numprocs, tag;
    int buffer;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    if(myid == 0){
        buffer = 5678;
        MPI_Isend(&buffer, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);
    }
    if(myid == 1){
        MPI_Irecv(&buffer, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
    }
    MPI_Wait(&request, &status);
    if(myid == 0){
        printf("Process %d  sent %d\n", myid, buffer);
    }
    if(myid == 1){
        printf("Process %d  got %d\n", myid, buffer);
    }
    MPI_Finalize();
}
```

Process 0 sends an integer to process 1.

Both processes use non-blocking message passing.

Blocking and Non-Blocking

- A non-blocking communication operation can be matched with a corresponding blocking operation.

- For example, a process sends a message using a non-blocking send operation, and the message can be received by the other process using a blocking receive operation.

Derived Datatypes

- Previously, MPI programs only exchange data which are stored as a continuous array and all data elements in a message must have the same type.

- Motivations
  1. In today's distributed memory systems, communication is much more expensive than local computation.

  2. The cost of sending a fixed amount of data in multiple messages is much greater than the cost of sending a single message with the same amount of data

- In MPI, derived datatype is an approach to consolidating multiple data into a single message, even if they are not stored continuously in memory.
  - Once you created a derived datatype, you can use it just like other basic MPI datatypes.

Derived Datatypes

- A derived datatype is used to represent a collection of data items in memory by storing both the types of the items and their relative locations in memory.
  - Idea: if a function that sends data knows the information about a collection of data items, it can collect the items from memory before they are sent.
  - Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.

Example of Derived Datatypes

- Assume a process wants to broadcast the following three variables
  - double a, b;
  - int n.
- A simple solution: call MPI_Bcast three times.
- Or, we can create a new datatype that can consolidate a, b, n into a single message, and then call a single MPI_Bcast.
  - Remember that data types in MPI have the type MPI_Datatype. So we are trying to create a variable with type MPI_Datatype.
- Formally, the derived datatype consists of a sequence of basic MPI data types together with a displacement for each of the data types.
  - Each data type is called a block, which can have multiple data elements.

Variable	Address
a	24
b	40
n	48

displacement

0

16

24

In the left example, the derived datatype consists of three blocks, each with a single data element.

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$$

MPI_Type_create_struct

- MPI_Type_create_struct() builds a derived datatype that consists of individual elements that have different basic types.

- int MPI_Type_create_struct(

```
int          count, /* in */
int          array_of_blocklengths[], /* in */
MPI_Aint          array_of_displacements[], /* in */
MPI_Datatype   array_of_types[], /* in */
MPI_Datatype*  new_type_p /* out */)
```

  - count is the number of blocks in the new datatype
  - array_of_blocklengths[] stores the number of data elements in each block
  - array_of_displacement[] stores the displacements (in bytes) of each block
  - array_of_types[] stores the MPI datatypes of elements in each block
  - new_type_p is a pointer to the MPI derived datatype

MPI_Get_address

- How to find out the displacements?
  - Get the addresses of the memory location of each element by MPI_Get_address();
  - Calculate the displacements by subtractions

- A special MPI type MPI_Aint is used to store memory address

- int MPI_Get_address(

```
        void*        location_p,  /* in */
        MPI_Aint*    address_p /* out */)
```

  - location_p is the pointer to the target variable
  - address_p is the variable used to store the memory address of the target variable

MPI_Type_commit and MPI_Type_free

- A derived datatype must be committed by MPI_Type_commit() before it can be used.
  - MPI_Type_commit() allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

- int MPI_Type_commit(MPI_Datatype* new_type_p);

- After using the new datatype, we can free its additional storage with MPI_Type_free()

- int MPI_Type_free(MPI_Datatype* old_type_p);

Derived Datatype: Example

```
1 #include "mpi.h"
2 #include <stdio.h>
3 struct mystruct {
4     char x;
5     double y[6];
6     int z[4];
7 };
8
9 int main(int argc, char *argv[]) {
10     struct mystruct mydata[1000];
11     int i, j, myid;
12     MPI_Status status;
13     MPI_Datatype mytype;
14     MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
15     int blocklen[3] = {1, 6, 4};
16     MPI_Aint disp[3];
17     MPI_Init(&argc, &argv);
18     disp[0] = &mydata[0].x - &mydata[0];
19     disp[1] = &mydata[0].y - &mydata[0];
20     disp[2] = &mydata[0].z - &mydata[0];
21     MPI_Type_create_struct(3, blocklen, disp, type, &mytype);
22     MPI_Type_commit(&mytype);
23     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

C/C++ structure for a single data element.
We will convert it into an MPI derived datatype.

Derived Datatype: Example

```
    if (myid == 0) {
        /* the code of initializing mydata[] should be put here */
        MPI_Send(mydata, 1000, mytype, 1, 0, MPI_COMM_WORLD);
    } else if (myid == 1){
        MPI_Recv(mydata, 1000, mytype, 0, 0, MPI_COMM_WORLD, &status);
    }
    MPI_Type_free (mytype);
    MPI_Finalize();
    return 0;
}
```

Collective Communication

- Point-to-point communication happens between two processes.

- Collective communication is a method of communication which involves participation of all processes in a communicator.
  - Example 1: process 0 needs to broadcast a vector to all other processes
  - Example 2: process 0 needs to receive data from all other processes and then perform a calculation

Motivation

- Consider the following example: how to sum up 8 numbers owned by 8 different processes?
  - Solution 1:

```
for( i = 1; i < 8; i++)
    Process i sends its number to process 0;
```

Process 0 adds the received 7 numbers with its own number.

  - Analysis:
    - Solution 1 involves 7 network transmissions, all with Process 0 as the destination. In a typical cluster, these 7 network transmissions take place sequentially (assuming the host of process 0 has a single network interface).

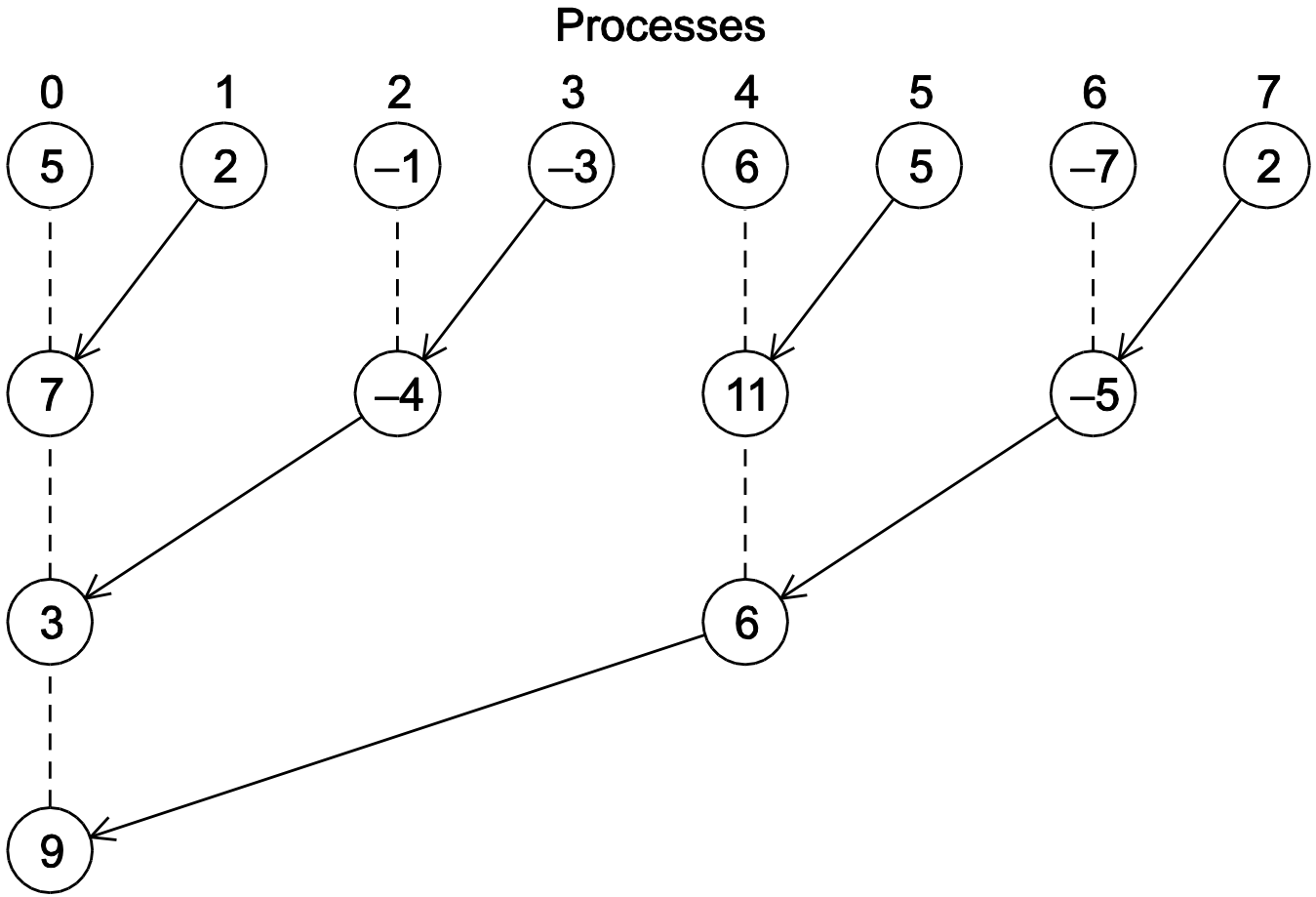
Motivation (Cont.)

- Solution 2 (a better one)
  - (1.a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
  - (1.b) Processes 0, 2, 4, and 6 add in the received values.

  - (2.a) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
  - (2.b) Processes 0 and 4 add the received values into their new values.

  - (3.a) Process 4 sends its newest value to process 0.
  - (3.b) Process 0 adds the received value to its newest value.

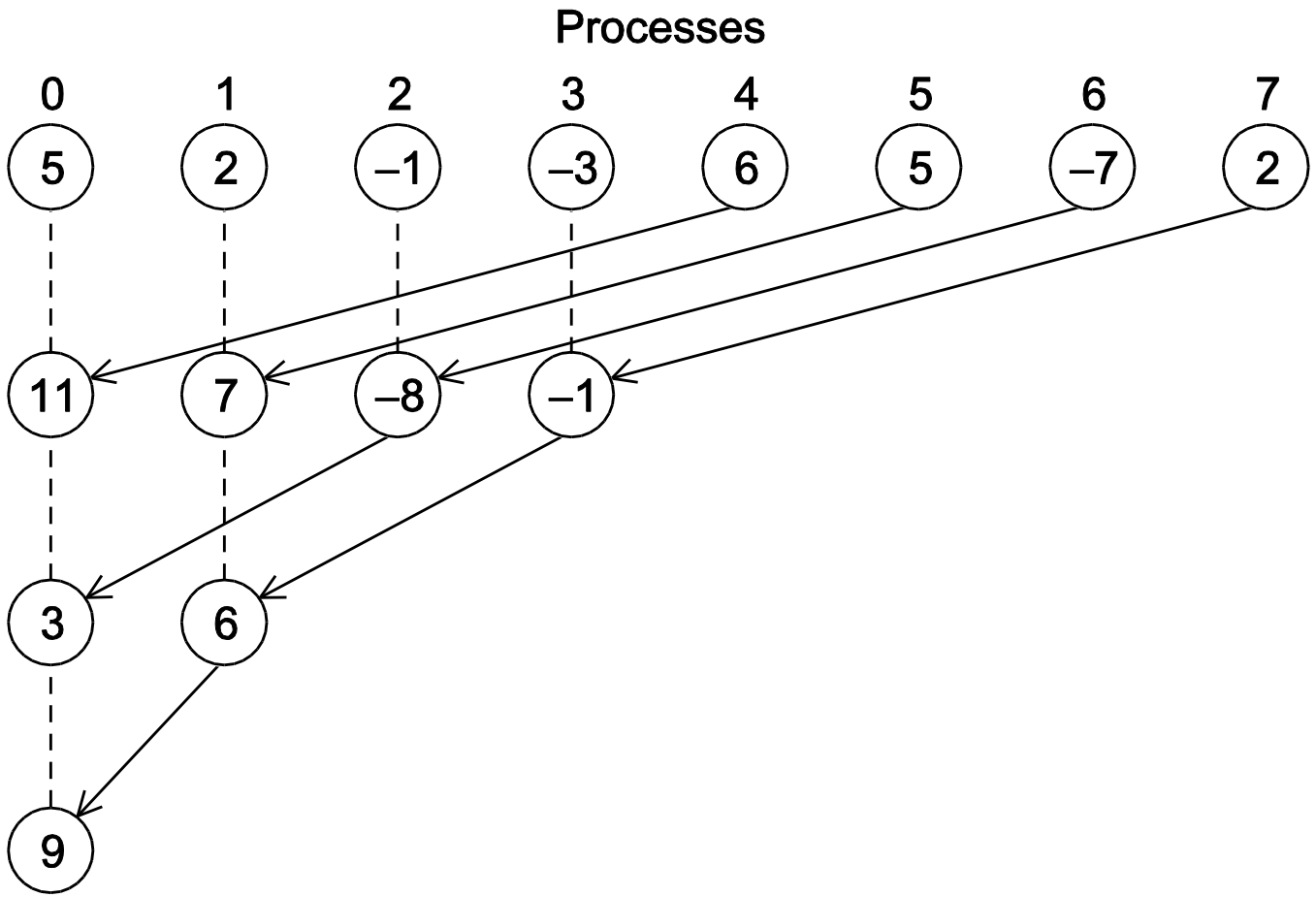
Solution 2: A Tree-Structured Global Sum



Analysis of Solution 2

- By using a network switch, multiple data communications can happen simultaneously if there is no common source/destination
- Solution 2 also needs a total of 7 data communications
- But, these 7 communications can happen in just 3 steps
  - Because some communications can take place in parallel
- It's easy to see that, summing up n numbers takes $\log_2 n$ steps only. Solution 1 takes n-1 steps.

An Alternative Tree-Structured Global Sum



Data Reduction in MPI

- The previous example is a typical pattern called "reduction", i.e., to reduce a set of messages into a single message within a process group.
  - Good solutions are much better than a simple solution.
  - There are different ways to implement a good solution.
  - A message can include multiple data elements. The reduction operation is applied element-wise on each data element.

MPI_Reduce

- MPI provides an optimized library function MPI_Reduce for reduction operation. The reduced results will be stored in a "destination process".

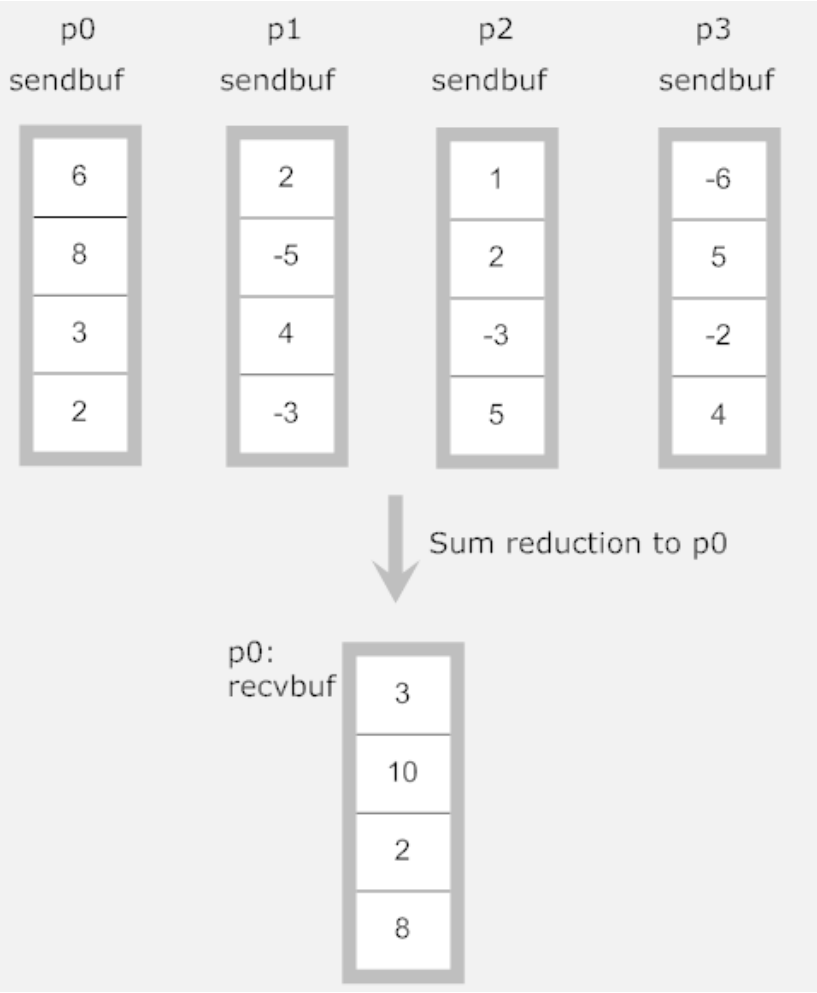
- int MPI_Reduce(void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op operator,
  int root,
  MPI_Comm comm)

  - <sendbuf, count, datatype> specify the "message"
  - <recvbuf, root> specify where to store the reduced result (i.e., recvbuf at process root)
  - operator specify the reduction operation (see next slide)
  - comm specifies the process group

MPI_Reduce

- The reduction operations are performed element-wise.

- Can recvbuf be the same as sendbuf at process 0?
  - Answer: NO!



MPI Reduction Operators

- The following operators are supported by MPI:

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Collective vs. Point-to-Point Communications

- All processes in the communicator must call the same collective function.

- For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.

Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be "compatible".

- For example, if one process passes in 0 as the root and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

Collective vs. Point-to-Point Communications

- The recvbuf argument is only used on dest_process.
- However, all of the processes still need to pass in an actual argument corresponding to recvbuf, even if it's just NULL.

Collective vs. Point-to-Point Communications

- Point-to-point communications are matched on the basis of tags and communicators.
- Collective communications don't use tags.
- They're matched solely on the basis of the communicator and the order in which they're called.

Test Yourself

- Suppose that each process calls MPI_Reduce with operator MPI_SUM, and destination process 0.
  - At the end, what will be the value of b and d at process 0, respectively?

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

Some Major Collective Communications

- MPI_Reduce
- MPI_Bcast
- MPI_Scatter
- MPI_Gather

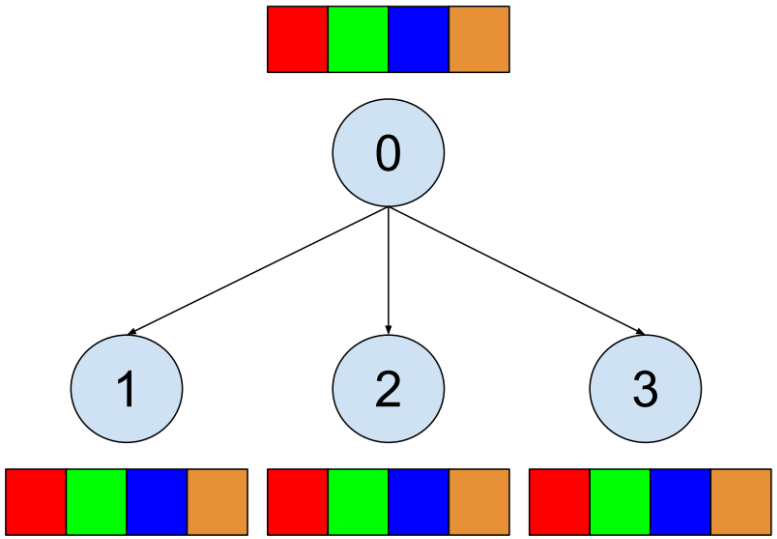
- MPI_Allgather
- MPI_Allreduce
- MPI_Alltoall
- MPI_Reduce_scatter

- MPI_Barrier

MPI_Bcast

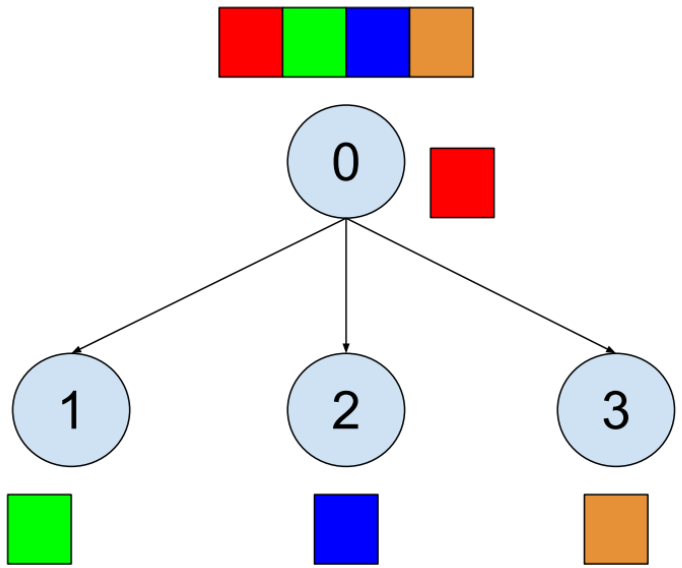
- Data belonging to a single process is sent to all of the processes in the process group.

- int MPI_Bcast(void *buf,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm comm)
  - <buf, count, datatype> specify the "message"
  - root specifies the rank of the source process
  - comm specifies the process group



MPI_Scatter

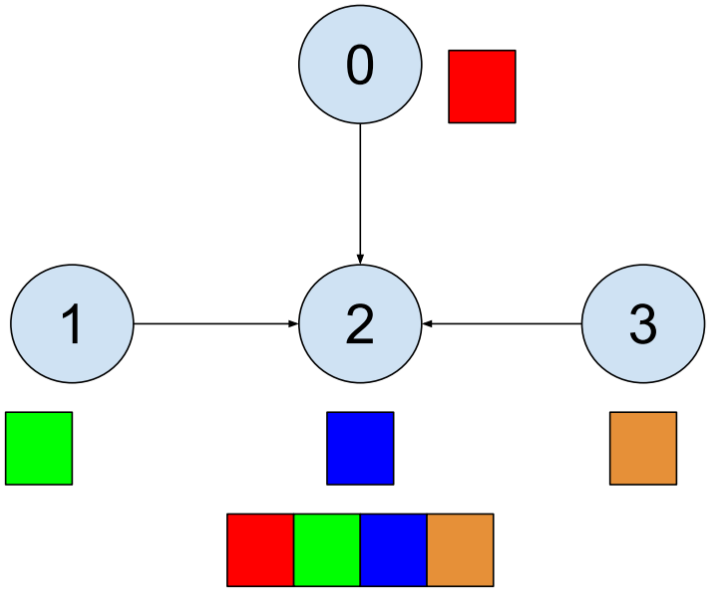
- MPI_Scatter evenly divides a vector of data owned by a root process into a number of pieces, and sends each piece to each of the other processes.
  - # of pieces equals # of processes in the process group
  - Assumption: the # of data items can be evenly divided by the # of processes



- int MPI_Scatter(void *sendbuf, int sendcount,
  MPI_Datatype senddatatype,
  void *recvbuf, int recvcount,
  MPI_Datatype recvdatatype,
  int root, MPI_Comm comm)
  - <sendbuf, sendcount, senddatatype> specify the "source message", where sendcount is the amount of data sent to each process, NOT the total amount of data in the sendbuf
  - <recvbuf, recvcount, recvdatatype> specify the "received message", where recvcount is the amount of data received by each process
  - root specifies the rank of the source process
  - comm specifies the process group

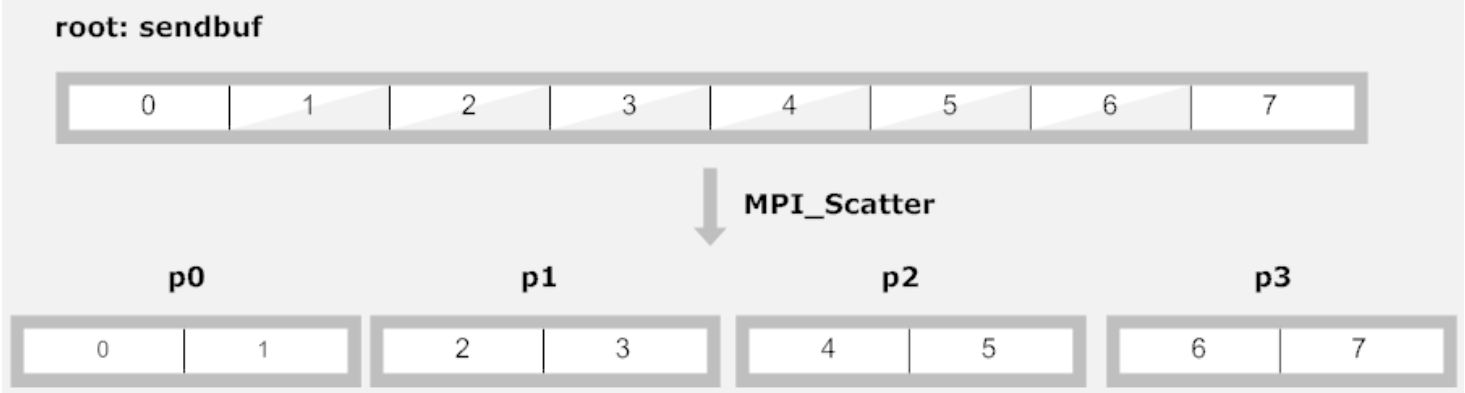
MPI_Gather

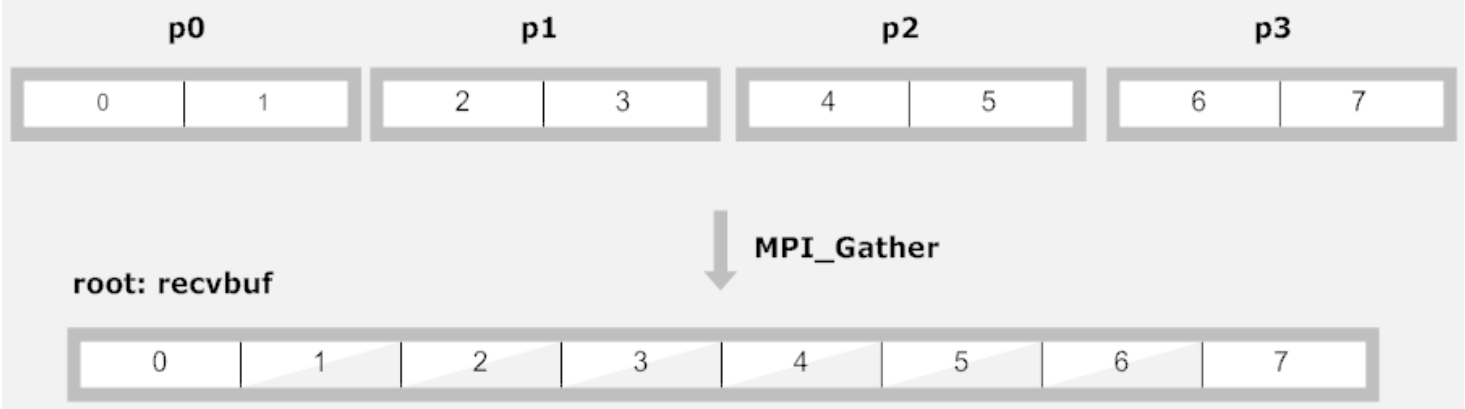
- MPI_Gather collects data from all processes in a process group onto a single process.
  - Remark: each process has to contribute the same amount of data items



- int MPI_Gather(void *sendbuf,
  int sendcount,
  MPI_Datatype senddatatype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvdatatype,
  int root,
  MPI_Comm comm)
  - <sendbuf, sendcount, senddatatype> specify the "source message", where sendcount is the amount of data sent out by each process
  - <recvbuf, recvcount, recvdatatype> specify the "received message", where recvcount is the amount of data received from **each process**
  - root specifies the rank of the destination process
  - comm specifies the process group

MPI_Scatter and MPI_Gather





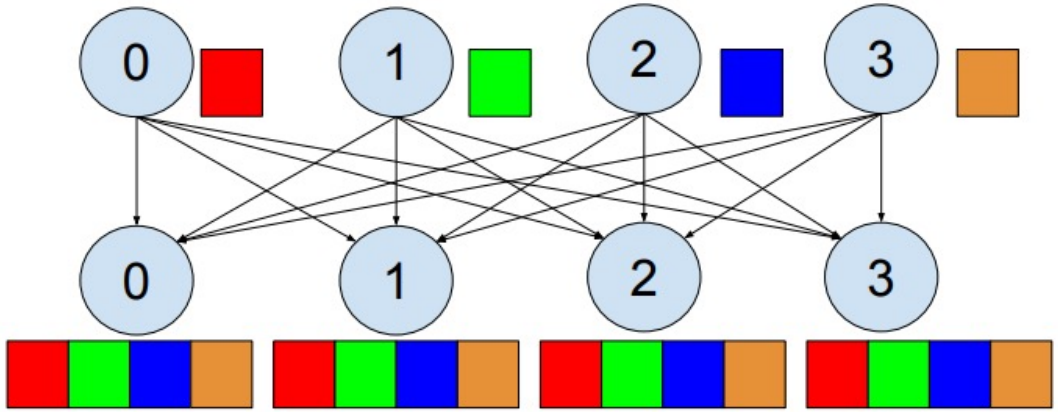
Example of MPI_Scatter: Distribute an Array

```
4     int n, local_n, my_rank, comm_sz;
5     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
6     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
7     if (my_rank == 0) {
8         scanf("%d", &n);
9         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
10    } else {
11        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
12    }
13    local_n = n / comm_sz;
14    local_a = malloc(local_n * sizeof(double));
15    if (my_rank == 0) {
16        a = malloc(n * sizeof(double));
17        for (i = 0; i < n; i++)
18            scanf("%lf", &a[i]);
19        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
20    } else {
21        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
22    }
```

MPI_Allgather

- MPI_Allgather concatenates the data of each process and stores the concatenated data in each process.

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

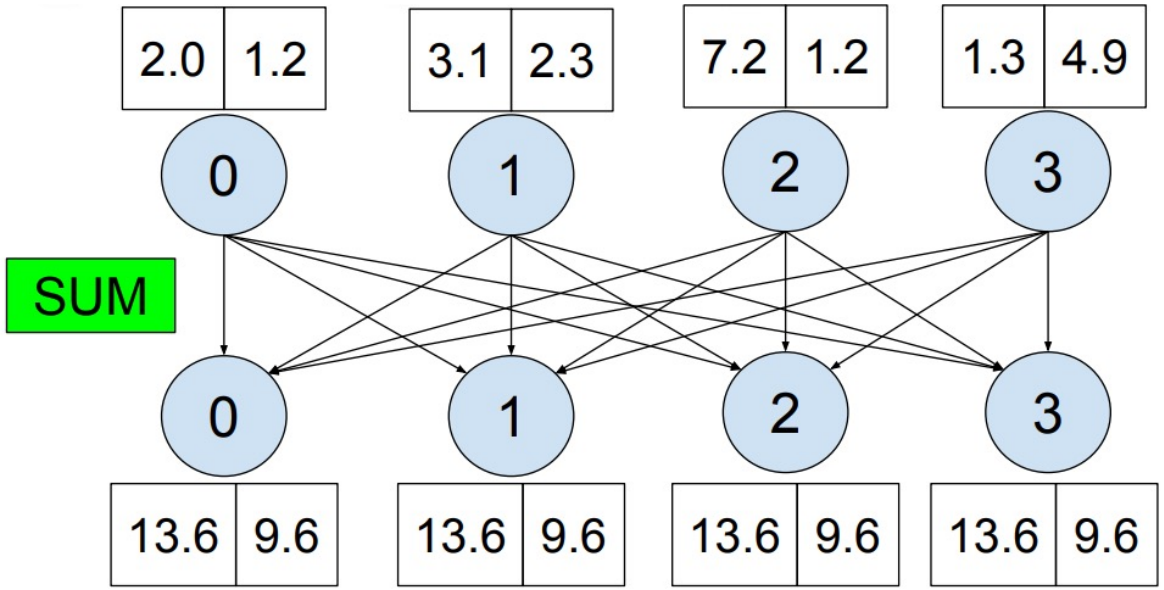


  - <sendbuf, sendcount, senddatatype> specify the "source message", where sendcount is the amount of data sent out by each process
  - <recvbuf, recvcount, recvdatatype> specify the "received message", where recvcount is the amount of data received from each process
  - comm specifies the process group

MPI_Allreduce

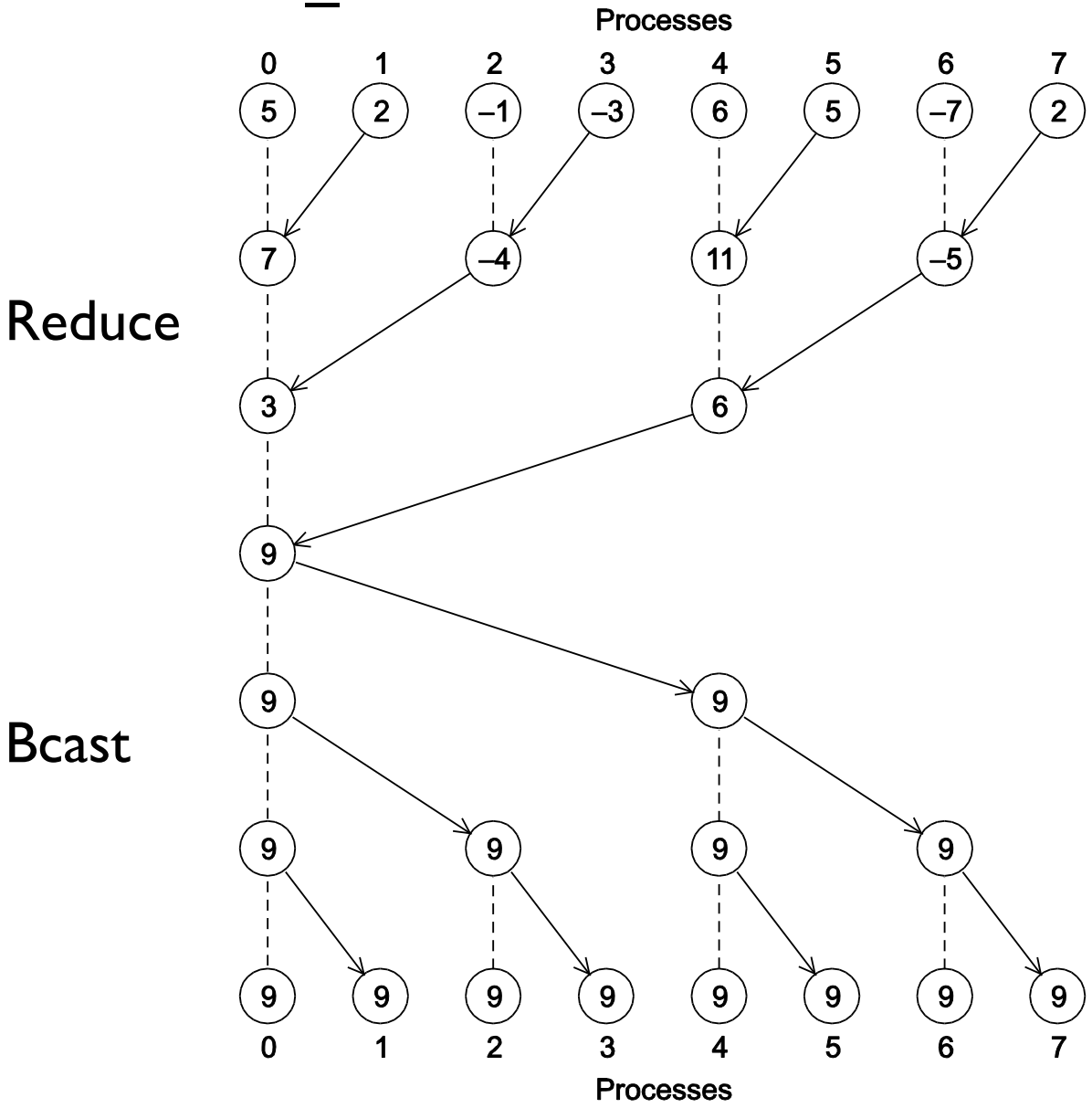
- int MPI_Allreduce(void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op operator,
  MPI_Comm comm)

- Similar to MPI_Reduce, but the reduced results are stored in ALL processes



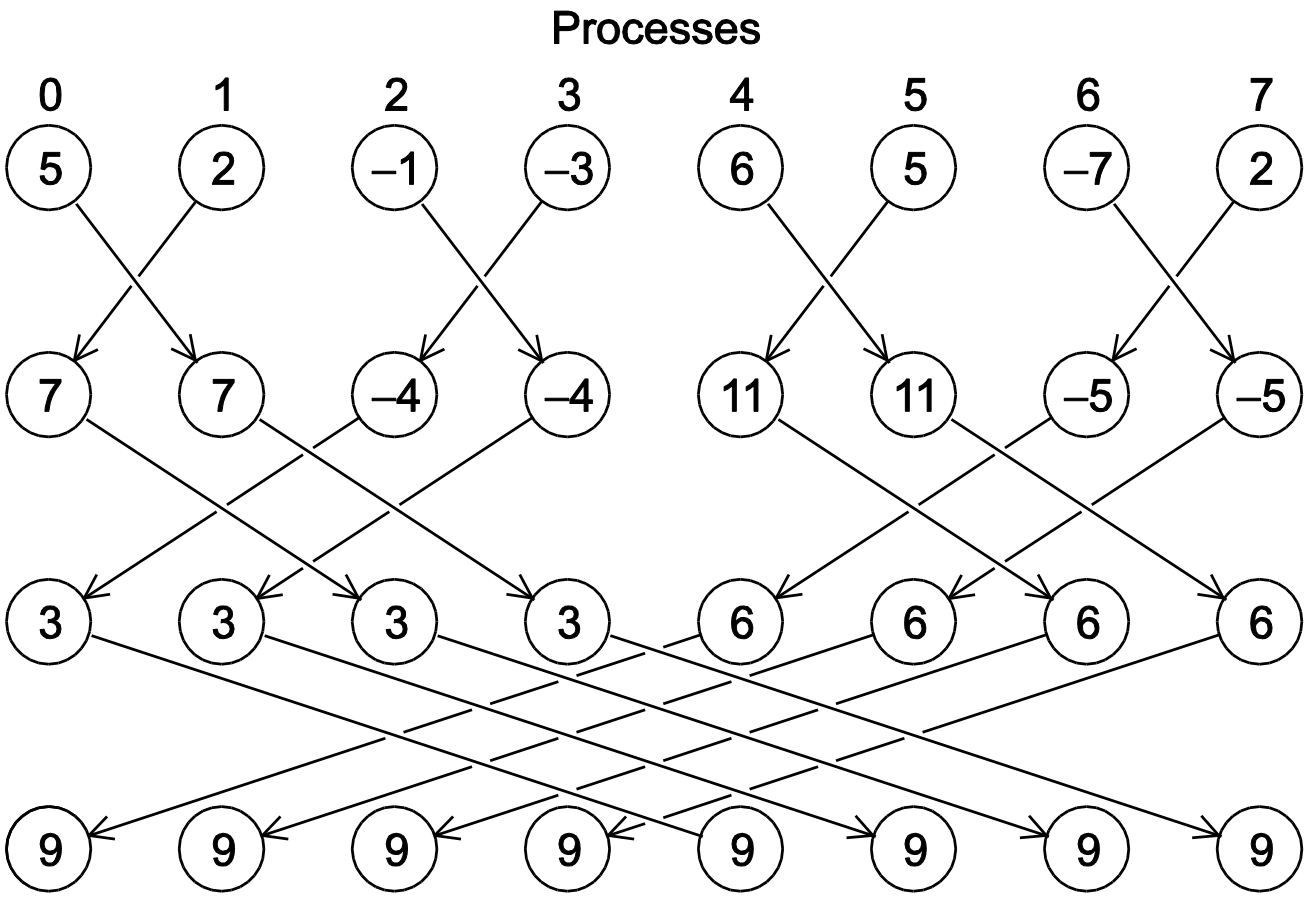
MPI_Allreduce: Algorithm 1

- A $2\log_2 n$ solution to MPI_Allreduce:



MPI_Allreduce: Algorithm 2

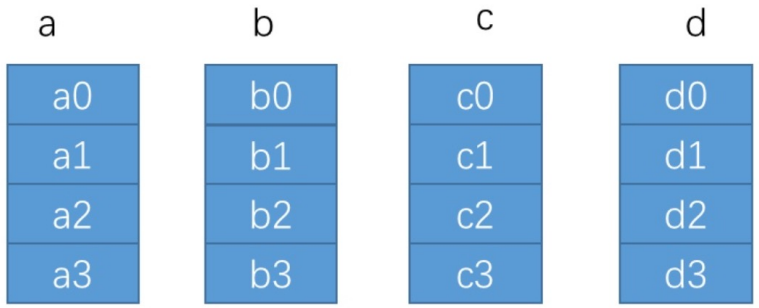
- A $\log_2 n$ solution to MPI_Allreduce: butterfly-structure
- It exploits the fact that today's network interface card and network switch support simultaneous send and receive operations.

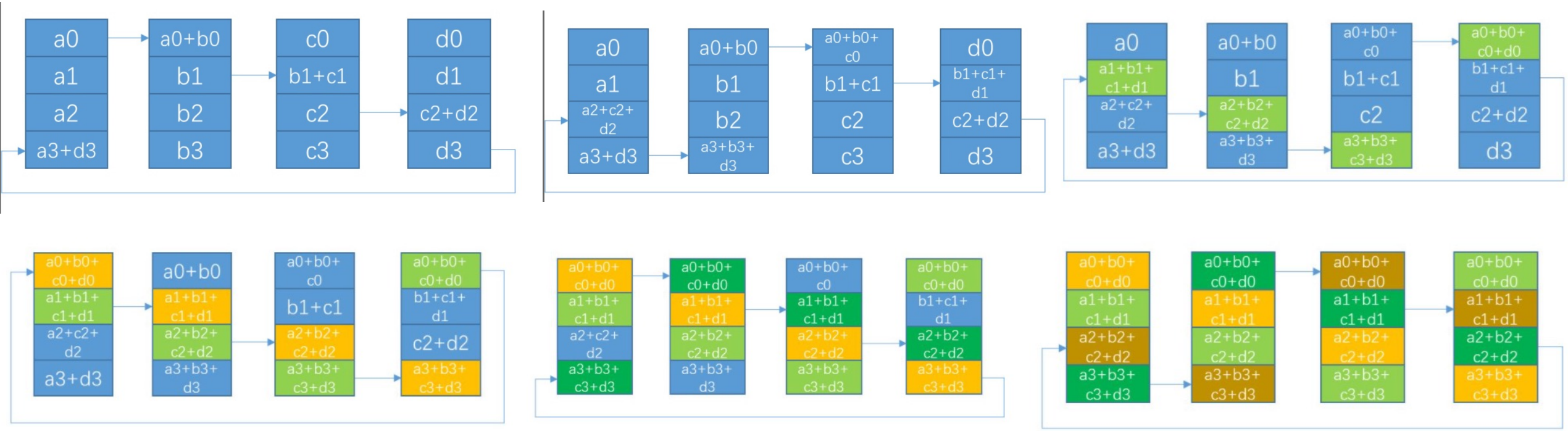


MPI_Allreduce: Algorithm 3

- A bandwidth friendly solution to MPI_Allreduce: ring-based

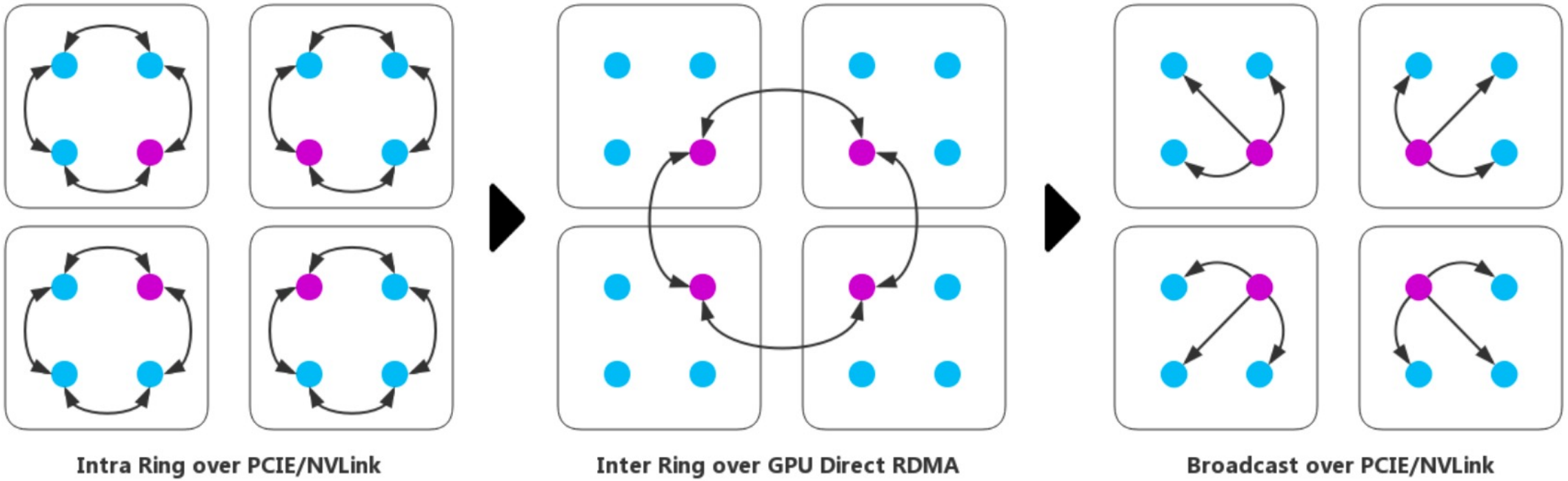
$$2(n-1)\alpha + 2(n-1)\beta m/n$$





Allreduce: Algorithm 4

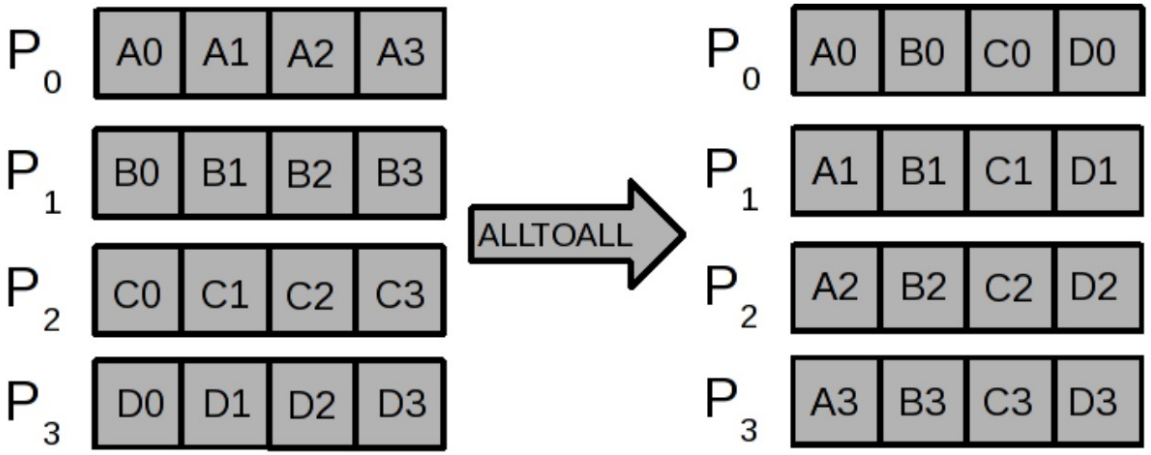
- A solution to balance the latency and bandwidth: hierarchical Allreduce [1]



[1] Jia et al., Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes, *NIPS Workshop 2018*

MPI_Alltoall

- int MPI_Alltoall (void *sendbuf,
  int sendcount,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvtype,
  MPI_Comm comm)



- Like a transpose
- Can be seen as a collection of simultaneous scatters or simultaneous gathers

MPI_Alltoall Example

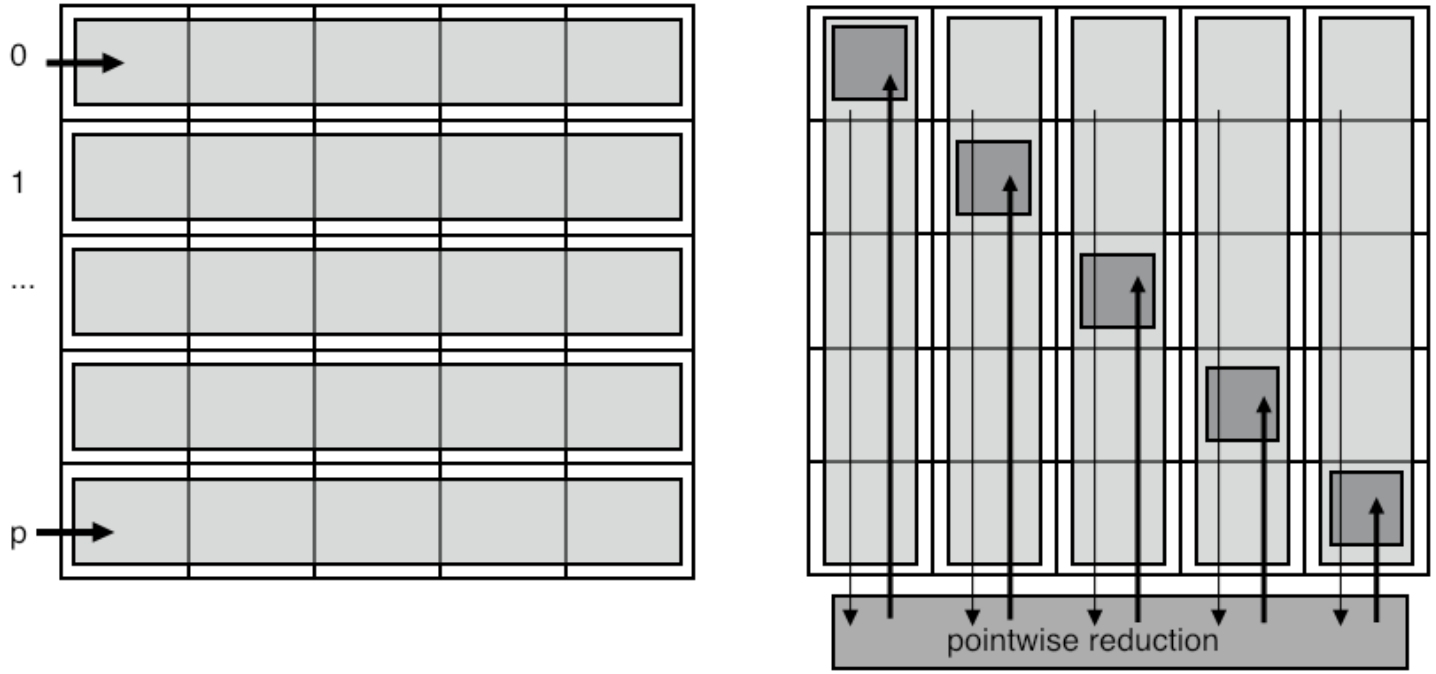
- Suppose there are four processes including the root, each with arrays u and v. After the all-to-all operation

MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_COMM_WORLD);

array u								Rank	array v							
10	11	12	13	14	15	16	17	0	10	11	20	21	30	31	40	41
20	21	22	23	24	25	26	27	1	12	13	22	23	32	33	42	43
30	31	32	33	34	35	36	37	2	14	15	24	25	34	35	44	45
40	41	42	43	44	45	46	47	3	16	17	26	27	36	37	46	47

MPI_Reduce_scatter

- int MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
  const int recvcounts[], MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm)
  - *sendbuf*: Starting address of send buffer.
  - *recvbuf*: Starting address of receive buffer (choice)
  - *recvcounts*: Integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.



MPI_Barrier

- Barrier is a commonly used synchronization method for a group of processes or threads.
  - Each member needs to wait at the barrier, until all members in the group have arrived at the barrier.

- MPI_Barrier implements the barrier synchronization operation.
  - The call to MPI_Barrier returns only after all the processes in the group have called this function.

- int MPI_Barrier(MPI_Comm comm)

Matrix-Vector Multiplication: Scatter and Gather

```c
int main(int argc, char** argv)
{
    int m = 0, n = 0, myid, numprocs, i;
    int srow = 0;
    double *A, *x, *y, *local_A, *local_y;
    double start, end;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if(myid == 0) {
        while ( m <= 0 || n <= 0 || m%numprocs != 0) {
            printf("Please input positive integers m and n: ");
            scanf("%d %d", &m, &n);
        }
        A = (double*) malloc( m * n * sizeof(double) );
        x = (double*) malloc(n * sizeof(double) );
        y = (double*) malloc(m * sizeof(double) );
        init_array(A, m * n);
        init_array(x, n);
    }
```

Prepare the data at Process 0

Matrix-Vector Multiplication: Scatter and Gather

```
27      MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
28      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
29
30      srow = m / numprocs;
31
32      local_A = (double*) malloc( srow * n * sizeof(double) );
33      local_y = (double*) malloc( srow * sizeof(double) );
34      if(myid != 0)     x = (double*) malloc(n * sizeof(double) );
35
36      MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
37
38      MPI_Scatter(A, srow * n, MPI_DOUBLE, local_A, srow * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
39
40      Mat_vect_mul(local_A, x, local_y, srow, n);
41
42      MPI_Gather(local_y, srow, MPI_DOUBLE, y, srow, MPI_DOUBLE, 0, MPI_COMM_WORLD);
43      if (myid == 0) { free(A); free(y); }
44      free(local_A); free(x); free(local_y);
45      MPI_Finalize();
46      return 0;
47  }
```

Broadcast m and n

Memory allocation

Scatter matrix A

Gather vector y

A Case Study

- Solving A System of Linear Equations by Gaussian Elimination

$$\mathbf{A\,x = b}$$

- It was used to evaluate the TOP500 supercomputers.
  - http://www.top500.org/project/linpack/

Linear Equations

- Consider the problem of solving the following linear equations:

$$
\begin{array}{ccccccccc}
a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} & = & b_0, \\
a_{1,0}x_0 & + & a_{1,1}x_1 & + & \cdots & + & a_{1,n-1}x_{n-1} & = & b_1, \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}.
\end{array}
$$

- This is written as $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is an $n \times n$ matrix with $\boldsymbol{A}[i, j] = a_{i,j}$, $\boldsymbol{b}$ is an $n \times 1$ vector $[\, b_0, b_1, \ldots, b_{n-1} \,]^T$, and $\boldsymbol{x}$ is the solution.

Solution: Gaussian Elimination

- Two steps: (1) reduction **A** to upper triangular form, and (2) back-substitution.

- The upper triangular form is as:

$$
\begin{array}{rcrcrcrcrcl}
x_0 & + & u_{0,1}x_1 & + & u_{0,2}x_2 & + & \cdots & + & u_{0,n-1}x_{n-1} & = & y_0, \\
 & & x_1 & + & u_{1,2}x_2 & + & \cdots & + & u_{1,n-1}x_{n-1} & = & y_1, \\
 & & & & & & & & \vdots & & \vdots \\
 & & & & & & & & x_{n-1} & = & y_{n-1}.
\end{array}
$$

- We write this as: $\boldsymbol{Ux} = \boldsymbol{y}$. Notice that we choose a **unit upper-triangular matrix** for $\boldsymbol{U}$, whose diagonal entries are all equal to one.

- A commonly used method for transforming a given matrix into an upper-triangular matrix is Gaussian Elimination.

Gaussian Elimination Example

$$
\begin{array}{rrrrcr}
4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\
2x_0 & & +5x_2 & -2x_3 & = & 4 \\
-4x_0 & -3x_1 & -5x_2 & +4x_3 & = & 1 \\
8x_0 & +18x_1 & -2x_2 & +3x_3 & = & 40
\end{array}
$$

Gaussian Elimination Example

$$\boxed{x_0 \quad +1.5x_1 \quad +0.5x_2 \quad -0.5x_3 \quad = \quad 2}$$

$$2x_0 \quad +5x_2 \quad -2x_3 \quad = \quad 4$$

$$-4x_0 \quad -3x_1 \quad -5x_2 \quad +4x_3 \quad = \quad 1$$

$$8x_0 \quad +18x_1 \quad -2x_2 \quad +3x_3 \quad = \quad 40$$

Gaussian Elimination Example

$$
\begin{array}{lllll}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
 & -3x_1 & +4x_2 & -1x_3 & = & 0 \\
 & +3x_1 & -3x_2 & +2x_3 & = & 9 \\
 & +6x_1 & -6x_2 & +7x_3 & = & 24
\end{array}
$$

Gaussian Elimination Example

$$
\begin{array}{lllll}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
 & \boxed{x_1 \quad -4/3x_2 \quad +1/3x_3 \quad = \quad 0} \\
 & +3x_1 & -3x_2 & +2x_3 & = & 9 \\
 & +6x_1 & -6x_2 & +7x_3 & = & 24
\end{array}
$$

Gaussian Elimination Example

$$
\begin{array}{ccccccc}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
 & x_1 & -4/3x_2 & +1/3x_3 & = & 0 \\
 & & 1x_2 & +1x_3 & = & 9 \\
 & & 2x_2 & +5x_3 & = & 24
\end{array}
$$

Gaussian Elimination Example

$$
\begin{array}{lllll}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
 & x_1 & -4/3x_2 & +1/3x_3 & = & 0 \\
 & & \boxed{x_2 \quad +x_3 \quad = \quad 9} & & & \\
 & & & 3x_3 & = & 6
\end{array}
$$

Gaussian Elimination Example

$$
\begin{array}{ccccccc}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
 & x_1 & -4/3x_2 & +1/3x_3 & = & 0 \\
 & & x_2 & +x_3 & = & 9 \\
 & & & x_3 & = & 2
\end{array}
$$

Back Substitution

$$x_3 = 2$$

$$x_2 = 9 - x_3 = 7$$

$$x_1 = 4/3x_2 - 1/3x_3 = 26/3$$

$$x_0 = 2 - 1.5x_1 - 0.5x_2 + 0.5x_3 = -27/2$$

Serial Gaussian Elimination

- A serial Gaussian Elimination algorithm that converts **Ax=b** to **Ux=y**

Input: ***A***, ***b***

Output: ***U***, ***y***

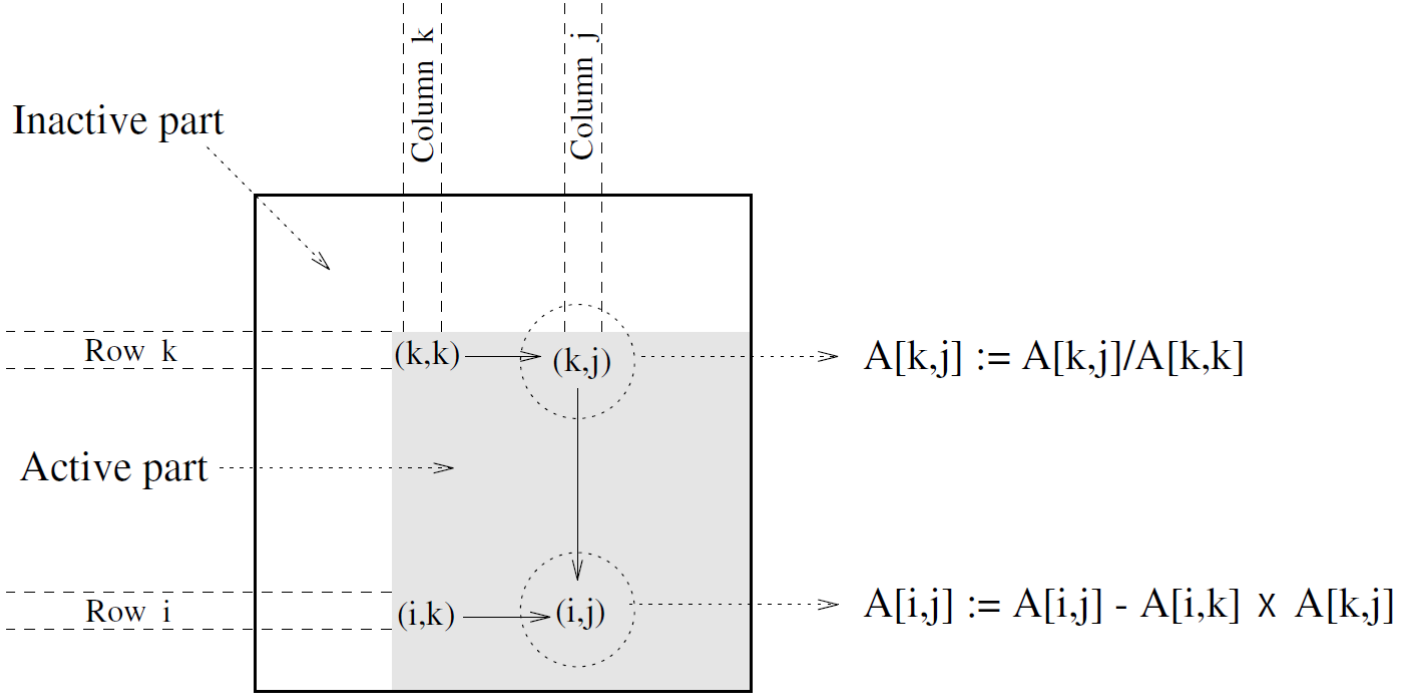
U is stored in the upper-triangular locations of ***A***.

```
1.  procedure GAUSSIAN_ELIMINATION (A, b, y)
2.  begin
3.      for k := 0 to n − 1 do                /* Outer loop */
4.      begin
5.          for j := k + 1 to n − 1 do
6.              A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.          y[k] := b[k]/A[k, k];
8.          A[k, k] := 1;
9.          for i := k + 1 to n − 1 do
10.         begin
11.             for j := k + 1 to n − 1 do
12.                 A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.             b[i] := b[i] − A[i, k] × y[k];
14.             A[i, k] := 0;
15.         endfor;            /* Line 9 */
16.     endfor;                /* Line 3 */
17. end GAUSSIAN_ELIMINATION
```

Source: Algorithm 8.4 of Ref. [1]

A Typical Computation in Gaussian Elimination

- The computation has three nested loops.
  - In the *k*th iteration of the outer loop, the algorithm performs $(n-k)^2$ computations.
  - Summing from $k = 1 \ldots n$, we have roughly $n^3/3$ multiplications-subtractions, or $2n^3/3$ operations.



Source: Figure 8.5 of Ref. [1]

Back-Substitution

- Sequential back-substitution algorithm for solving an upper-triangular system of equations Ux=y
  - ***U*** is a unit upper-triangular matrix (i.e., ***U***[*k*,*k*] = 1)
  - It takes $n^2/2$ multiplications and subtractions

```
procedure BACK_SUBSTITUTION (U, x, y)
begin
    for k := n − 1 downto 0 do   /* Main loop */
        begin
            x[k] := y[k];
            for i := k − 1 downto 0 do
                y[i] := y[i] − x[k] × U[i, k];
        endfor;
end BACK_SUBSTITUTION
```

Source: Algorithm 8.5 of Ref. [1]

A Small Test

- Tianhe-2 is one of the fastest supercomputers in the world (rank 6th as of 2020)
  - http://www.top500.org/system/177999

- Its Linkpack performance (i.e., R_{max}) is 33,862.7 TFlop/s, measured by solving a linear system with 9,960,000 (i.e., N_{max}) linear equations.

- Can you estimate how long does Tianhe-2 take to solve the linear system with N_{max} equations?

Partial Pivoting

- In the previous algorithm, we implicitly assume that A[k,k] is not zero or close to zero.
  - Why?

- Partial pivoting (by column exchange)
  - At the beginning of the outer loop in the *k*th iteration, select a column *i* such that $\boldsymbol{A}[k, i]$ is the largest in magnitude among all $\boldsymbol{A}[k, j]$, $k \leq j < n$. We call column *i* the **pivot column**.
  - Exchange the *k*th and the *i*th columns before starting the iteration.

- Partial pivoting (by row exchange)

Parallel Gaussian Elimination: 1-D Partitioning

- We can apply rowwise 1-D partitioning on the coefficient matrix A.

- Assume each row is assigned to a process
  - An n x n matrix requires n processes.
  - Process i (P_i) initially stores data $\boldsymbol{A}[i, j]$, $0 \leq j < n$

- How to parallelize Algorithm 8.4 shown in Page 68?

Parallel Gaussian Elimination: 1-D Partitioning

- In the outer loop (line 3), each iteration has a dependency on the previous iterations (in the general sense)
  - We cannot easily parallelize at this level

- In the inner level (lines 9-15), the elimination step for each row is independent from each other
  - These computations can be done in parallel by different processes

An Example: n = 8, k = 3

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.6 (a) of Ref. [1]

Lines 5-8:

```
for j := k+1 to n-1 do
    A[k,j] := A[k,j] / A[k,k];
y[k] := b[k] / A[k,k];
A[k,k] := 1;
```

The above codes are executed by Process P_3 only.

No communication is required.

An Example: n = 8, k = 3

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.6 (b) of Ref. [1]

After the calculations of lines 5-8, data ***A***[*k*, *j*], *k*<*j*<*n*, need to be broadcast to processes P_4 – P_7.

We call these data "**active data**".

An Example: n = 8, k = 3

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.6 (c) of Ref. [1]

Lines 9-15:

```
for i := k+1 to n-1 do
begin
   for j := k+1 to n-1 do
      A[i,j] -= A[i,k] * A[k,j];
   b[i] -= A[i,k] * y[k];
   A[i,k] := 0;
end
```

The above codes are executed by Processes P_4-P_7 in parallel.

No communication is required.

Analysis for the Case of n Processors

- The first step of the algorithm normalizes the row. This is a serial operation and takes time (*n*-*k*-1) in the *k*-th iteration.

- In the second step, the active data in the normalized row are broadcast to (n-k-1) processors. This takes time

$$(t_s + t_w(n-k-1))\log n$$

- Each processor can independently eliminate this row from its own. This requires (*n*-*k*-1) multiplications and subtractions.

- To summarize, the *k*-th iteration takes 3(*n*-*k*-1) calculations and

$$(t_s + t_w(n-k-1))\log n$$

communications.

Analysis for the Case of n Processors

- The total parallel run time can be computed by summing from $k = 0 \ldots n\text{-}1$ as

$$T_P = \frac{3}{2}n(n-1) + t_s n \log n + \frac{1}{2}t_w n(n-1)\log n.$$

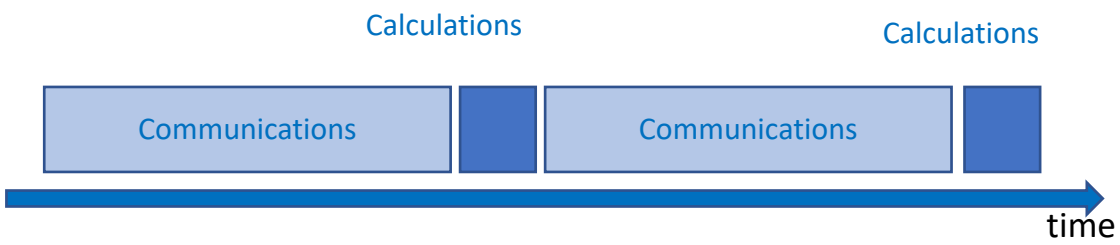
- The cost of the parallel solution is nT_p, which is

$$\Theta(n^3 \log n)$$

- So this solution is not cost-optimal!

Two Inefficiencies

- Communications overhead
  - For each iteration k, each process performs $\Theta(n\text{-}k)$ calculations in parallel. But the communication time for each iteration is $\Theta(n\text{-}k)\log_2 n$!



- Load imbalance
  - Process P_i will become totally idle after the $(i+1)$-th iteration
  - Process P_o has the lightest workload of $\Theta(n)$, while $P_{n\text{-}1}$ has the heaviest workload of $\Theta(n^2)$.

How to Improve?

- In the previous method, the (k+1)-th iteration starts only after all the computation and communication for the k-th iteration is complete.
  - Is it a must?
- Inside each iteration of k, the calculations happen after the data communications.
  - Is it a must?
- To answer the above questions, we need to have an in-depth analysis of serial Algorithm 8.4.

In-depth Analysis of Algorithm 8.4

- In iteration k, Process P_k broadcasts its active data to Processes $P_{k+1}, \ldots, P_{n-1}$.
  - This can be done by a sequential forwarding: $P_k \rightarrow P_{k+1}, P_{k+1} \rightarrow P_{k+2}, \ldots, P_{n-2} \rightarrow P_{n-1}$.

- For process P_{k+1}, after forwarding the active data to P_{k+2}, it need not wait for all processes to receive the active data. Instead, it can start to perform the elimination step (Line 12)!
  - We can overlap communications with computations!

- After P_{k+1} performs its elimination step in the k-th iteration, it can start to perform the division step (Line 6) for the (k+1)-th iteration. Then it can start to send its active data to process P_{k+2}.
  - We can overlap iteration K+1 with iteration k! It can further overlap communications with computations, and also simultaneous communications!

- Next, we will show a "pipelined" parallel solution by a 5x5 example.

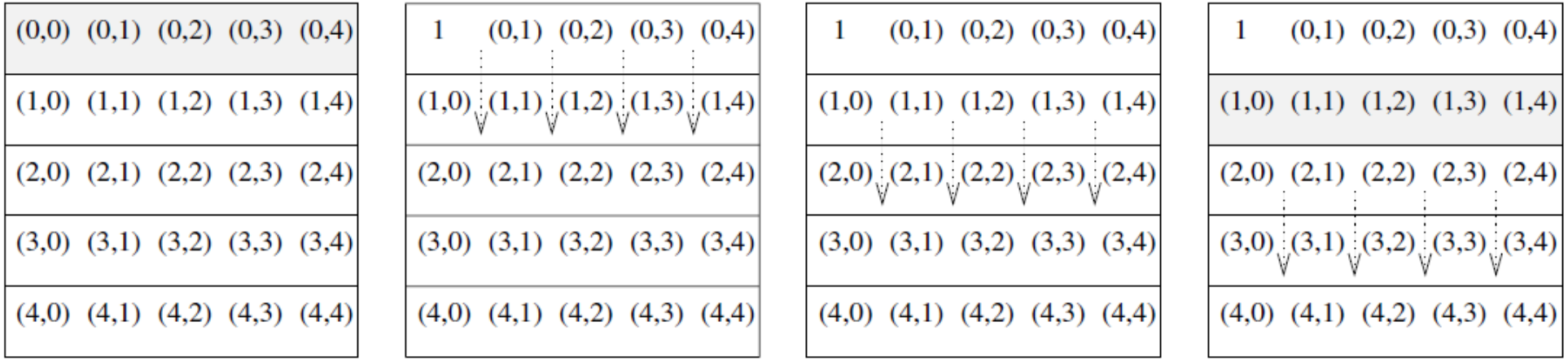
An Example of 5x5 Matrix

(a) P_0 performs the division on row 0.

(b) P_0 forwards its active data to P_1 .

(c) P_1 forwards its active data to P_2 .

(d) P_1 performs elimination. Meanwhile, P_2 forwards its active data to P_3.



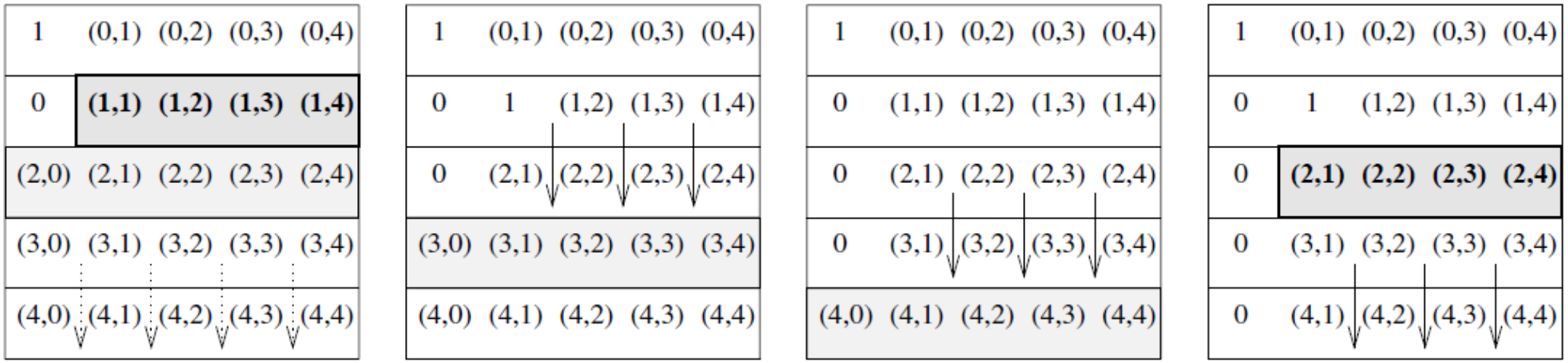
(a) Iteration k = 0 starts (b) (c) (d)

Source: Figure 8.7 (a-d) of Ref. [1]

An Example of 5x5 Matrix

(e) P_2 performs the elimination. Meanwhile, P_3 forwards its active data to P_4. Furthermore, P_1 performs the division on row 1 – iteration k=1 starts while iteration k=0 has not finished yet!

(f-p) Go through them by yourselves.



(e) Iteration k = 1 starts (f) (g) Iteration k = 0 ends (h)

Source: Figure 8.7 (e-h) of Ref. [1]

An Example of 5x5 Matrix

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	**(2,2)**	**(2,3)**	**(2,4)**
0	**(3,1)**	**(3,2)**	**(3,3)**	**(3,4)**
0	(4,1)	(4,2)	(4,3)	(4,4)

(i) Iteration k = 2 starts

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	(3,2)	(3,3)	(3,4)
0	**(4,1)**	**(4,2)**	**(4,3)**	**(4,4)**

(j) Iteration k = 1 ends

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	(3,2)	(3,3)	(3,4)
0	0	(4,2)	(4,3)	(4,4)

(k)

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	**(3,2)**	**(3,3)**	**(3,4)**
0	0	(4,2)	(4,3)	(4,4)

(l)

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	0	(3,3)	(3,4)
0	0	**(4,2)**	**(4,3)**	**(4,4)**

(m) Iteration k = 3 starts

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	0	1	(3,4)
0	0	0	(4,3)	(4,4)

(n)

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	0	1	(3,4)
0	0	0	(4,3)	(4,4)

(o) Iteration k = 3 ends

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	0	1	(2,3)	(2,4)
0	0	0	1	(3,4)
0	0	0	0	**(4,4)**

(p) Iteration k = 4

Source: Figure 8.7 (i-p) of Ref. [1]

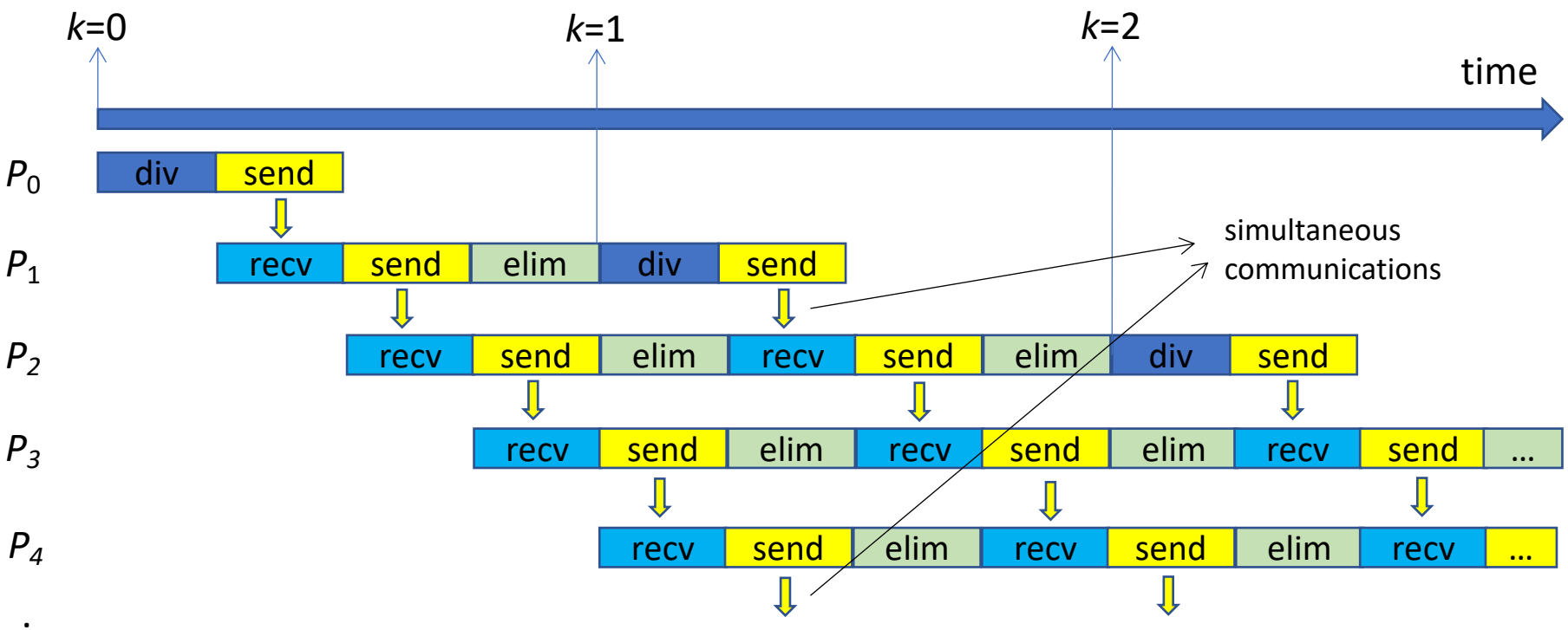
Pipelined Parallel Solution

- In the pipelined version, each process P_i performs the followings:

```
step = 0;
while (step < i) {
   Receive the "active data" from P_{i-1};
   Send the "active data" to P_{i+1} ;
   Perform the elimination ;
   step++;
}
perform the division;
send the active data to P_{i+1};
```

Analysis of Pipelined Solution

Remark: We simply assume div, recv, send, elim take the same time.



div: division step
send: send data to next process
recv: receive data from the previous process
elim: elimination step

Analysis of Pipelined Solution

- From the previous pipeline diagram, it is obvious that the time interval between two iterations is $\Theta(n)$.
- There is a total of n iterations. So the last iteration starts from $\Theta(n^2)$.
- The last iteration takes $\Theta(1)$ only.
- So the parallel run time is $\Theta(n^2)$, which is cost optimal.

Back to Reality: p < n

- Previously, we assume the number of processes (or processors) $p = n$
- In most scenarios, the number of processors p is much smaller than n
- Block 1-D rowwise partition: each process is assigned n/p contiguous rows of the matrix

Example of Block 1-D Partitioning

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_2	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_3	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.8 of Ref. [1]

In the k=3 iteration, the active data on P_1 need to be sent to P_2 and P_3.

As compared with the case of $p = n$, the communication cost is reduced by a factor of n/p.

When $n >> p$, computations dominate communications.

Load Balancing Issue

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_2	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_3	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Block 1-D mapping

Source: Figure 8.9 (a) of Ref. [1]

The cost of the block 1-D partitioning is n^3, while the cost of the serial algorithm is $2n^3/3$.

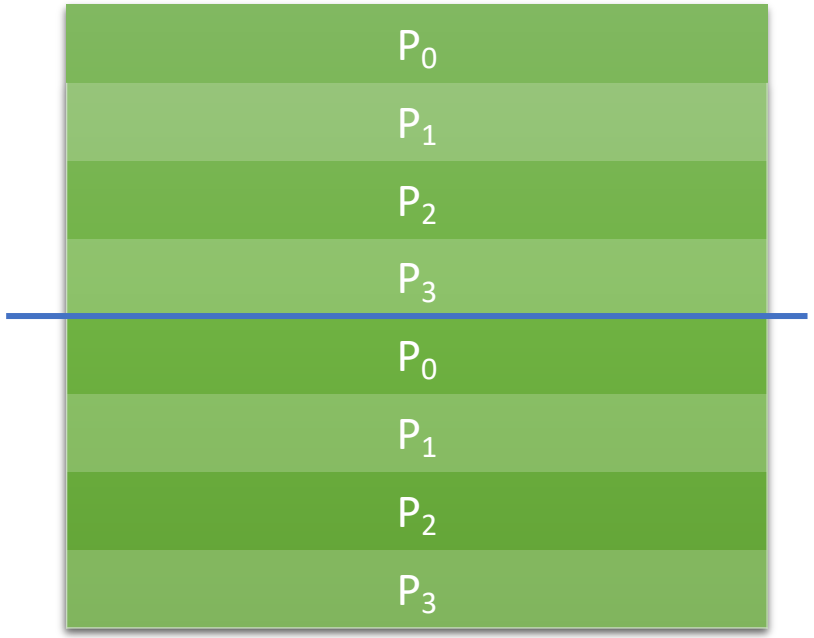
This is caused by the uneven load distribution among p processes.

In the left example, P_0 is completely idle, P_1 is partially loaded, P_2 and P_3 are fully loaded.

Cyclic 1-D Mapping

- **Cyclic mapping** can be used to improve load balancing.

- In each iteration, the computations are distributed to all processes.



1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	P_0
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	P_1
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	P_2
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	P_3
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	

(b) Cyclic 1-D mapping

Source: Figure 8.9 (a) of Ref. [1]

References

1. Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta, Introduction to Parallel Computing, 2nd Edition, Addison Wesley, 2003.

2. Peter S. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2010.

3. William Gropp, Ewing Lush, Anthony Skjellum, Using MPI, 2nd Edition, The MIT Press, 1999.

4. C. Renggli et al., SparCML: High-Performance Sparse Communication for Machine Learning, SC, 2019