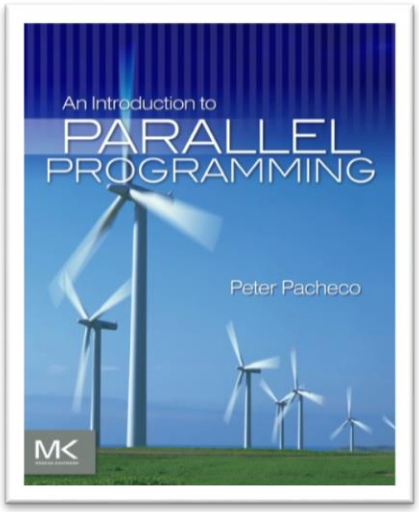
高性能计算程序设计基础

任课教师：黄聃（Huang，Dan）



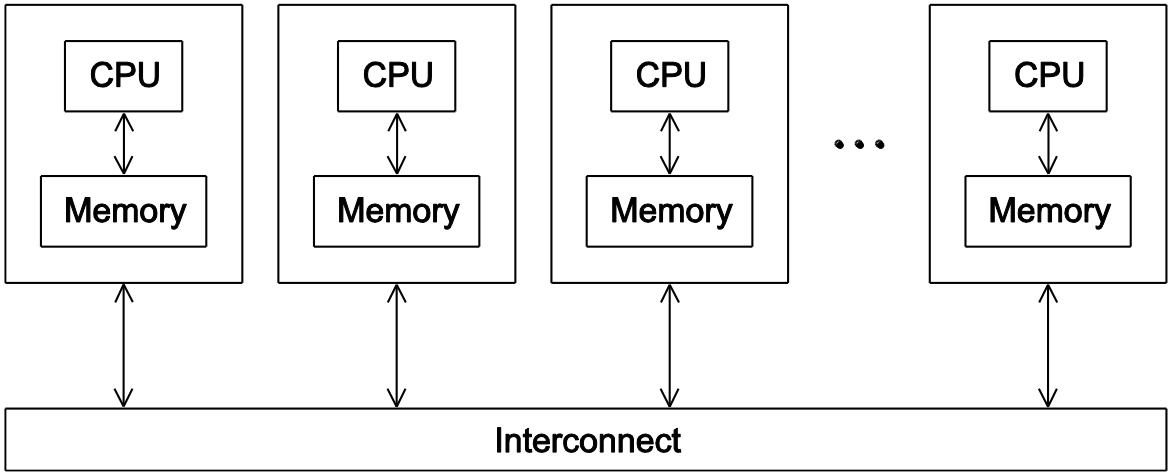
Chapter 3

Distributed Memory Programming with MPI

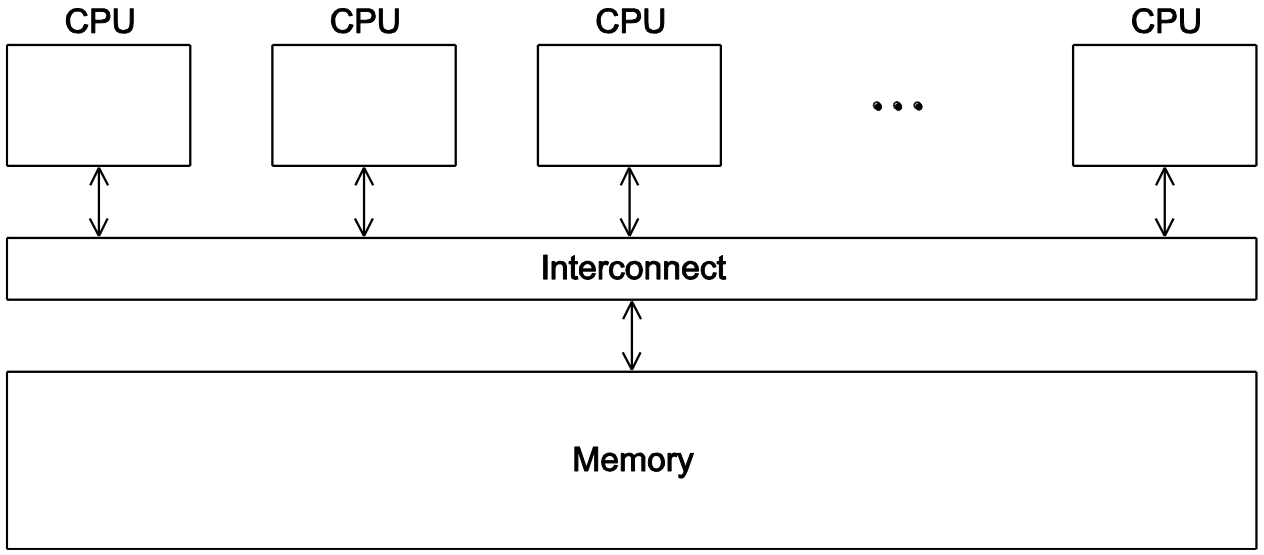
Roadmap

- Writing your first MPI program.
- Using the common MPI functions.
- The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.

A distributed memory system



A shared memory system



Hello World!

```
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

(a classic)

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - MPI_Comm_size reports the number of processes.
  - MPI_Comm_rank reports the rank, a number between 0 and size-1, identifying the calling process

Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.

- *p* processes are numbered *0, 1, 2, .. p-1*

Our first MPI program

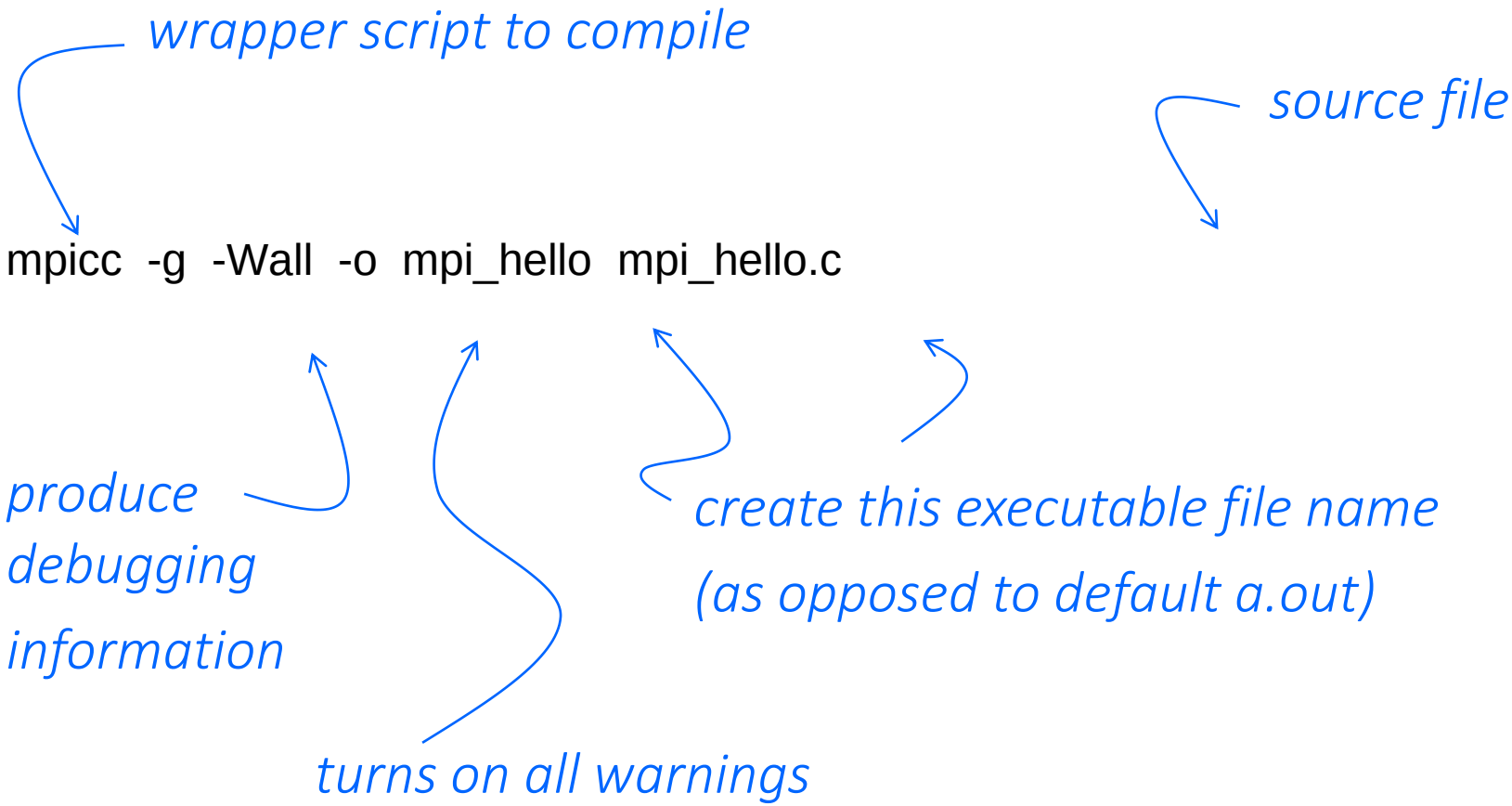
```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char       greeting[MAX_STRING];
9     int        comm_sz;  /* Number of processes */
10    int        my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

Set MPICH2 Path

- Then add the following at
- the end of your ~/.bashrc file and source it with the command
- source ~/.bashrc (or log in again).
- #--
- # MPICH2 setup
- export PATH=/opt/MPICH2/bin:$PATH
- export MANPATH= =/opt/MPICH2/bin :$MANPATH
- export LD_LIBRARY_PATH=/opt/MPICH2/bin:$LD_LIBRARY_PATH
- #--

- Some logging and visulaization help:
- You can Link with the libraries -llmpe -lmpe to enable logging and the MPE environment. Then run the program as usual and a log file will be produced. The log file can be visualized using the jumpshot program that comes bundled with MPICH2.

Compilation



Setup Multiple Computer Environment

Create a file called say “machines” containing the list of machines:

athena.cs.siu.edu
oscarnode1.cs.siu.edu

…….

oscarnode8.cs.siu.edu

~~Establish network **environments**~~

~~mpdboot –n 9 –f machines~~

~~mpdtrace~~

~~mpdallexit~~

Execution

mpiexec -n <number of processes> <executable>

---

mpiexec -n 1 ./mpi_hello

run with 1 process

mpiexec -n 4 ./mpi_hello

run with 4 processes

Execution

```
mpiexec  -n  1  ./mpi_hello

    Greetings from process 0 of 1 !
```

```
mpiexec  -n  4  ./mpi_hello

    Greetings from process 0 of 4 !
    Greetings from process 1 of 4 !
    Greetings from process 2 of 4 !
    Greetings from process 3 of 4 !
```

MPI Programs

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with "MPI_".
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.

MPI Components

- MPI_Init
  - Tells MPI to do all the necessary setup.

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

- MPI_Finalize
  - Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

Basic Outline

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
    . . .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    . . .
    MPI_Finalize();
    /* No MPI calls after this */
    . . .
    return 0;
}
```

Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.
- Called MPI_COMM_WORLD.

Communicators

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);
```

number of processes in the communicator

```
int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

my rank
(the process making this call)

Sample MPI program

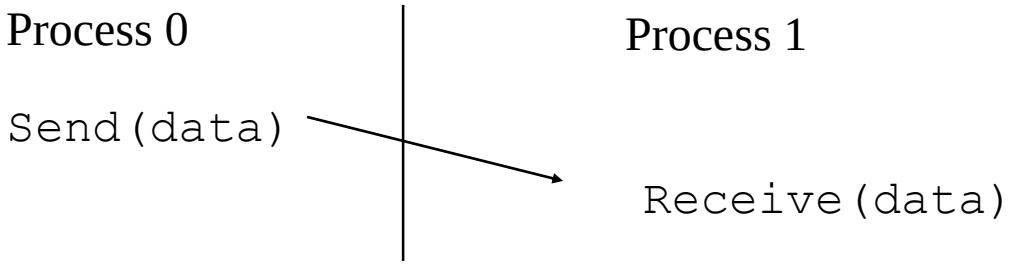
```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
        int myid, numprocs;
        int data[MAXSIZE], i, x, low, high, myresult, result;
        char fn[255];
        char *fp;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        if (myid == 0) {  /* Open input file and initialize data */
                strcpy(fn,getenv(“HOME”));
                strcat(fn,”/MPI/rand_data.txt”);
                if ((fp = fopen(fn,”r”)) == NULL) {
                        printf(“Can’t open the input file: %s\n\n”, fn);
                        exit(1);
                }
                for(i = 0; i < MAXSIZE; i++) fscanf(fp,”%d”, &data[i]);
        }
        MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
        x = n/nproc; /* Add my portion Of data */
        low = myid * x;
        high = low + x;
        for(i = low; i < high; i++)
                myresult += data[i];
        printf(“I got %d from %d\n”, myresult, myid); /* Compute global sum */
        MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0) printf(“The sum is %d.\n”, result);
        MPI_Finalize();
}
```

SPMD

- Single-Program Multiple-Data
- We compile one program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.
- The if-else construct makes our program SPMD.

MPI Basic Send/Receive

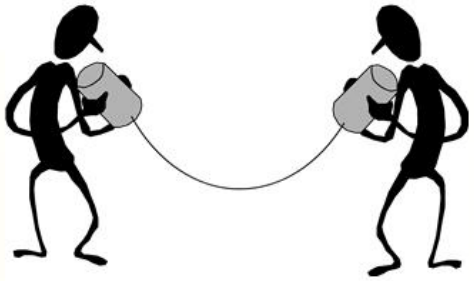
- We need to fill in the details in



- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Communication

```
int MPI_Send(
    void*           msg_buf_p      /* in */,
    int             msg_size       /* in */,
    MPI_Datatype    msg_type       /* in */,
    int             dest           /* in */,
    int             tag            /* in */,
    MPI_Comm        communicator   /* in */);
```



MPI Datatypes

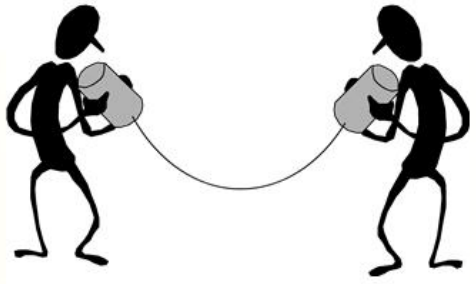
- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays
- May hurt performance if datatypes are complex

Data types

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long 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	

Communication

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```



MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI_ANY_TAG** as the tag in a receive
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

Count of data

r

MPI_Send

src = q

MPI_Recv

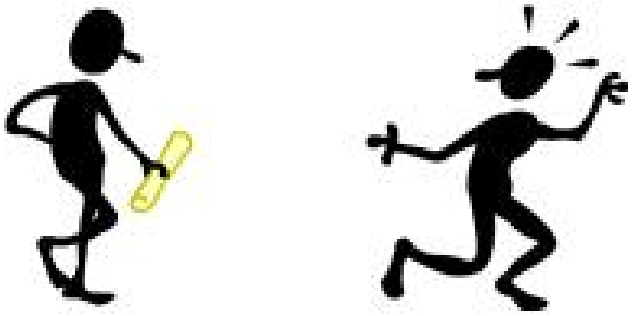
dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message, (MPI_ANY_SOURCE)
  - or the tag of the message. (MPI_ANY_TAG)



status_p argument

How receiver finds out the sender, tag if they are not needed by the receiver

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Count of received data

MPI_Status*

MPI_SOURCE

MPI_TAG

MPI_ERROR

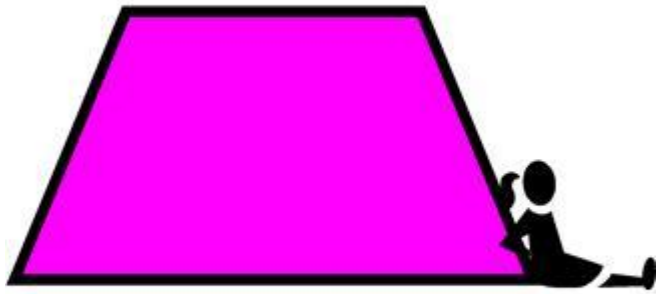
```
MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG
```

How much data am I receiving?

```
int MPI_Get_count(
          MPI_Status*   status_p   /* in  */,
          MPI_Datatype  type       /* in  */,
          int*          count_p    /* out */);
```

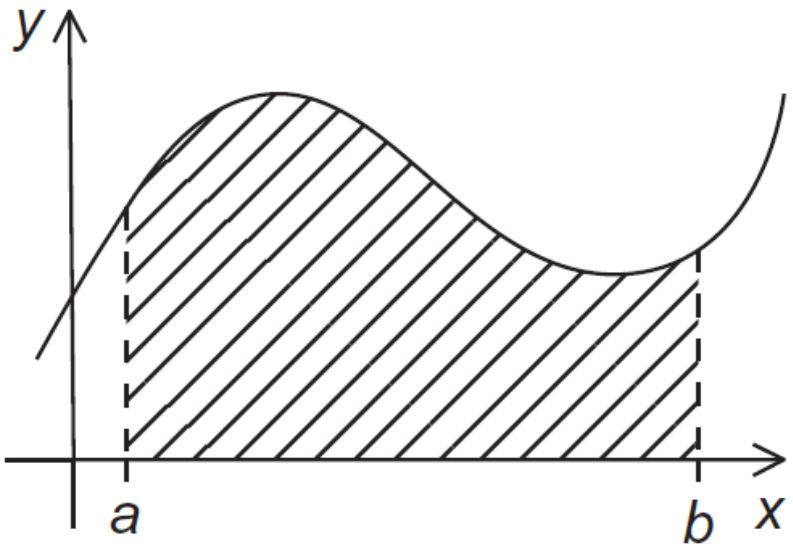
Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send (**locally blocking with buffer** copied to internal storage or **block** starts transmission) may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Non-blocking MPI_Isend and MPI_Irecv, immediate return.
- Know your implementation; don't make assumptions!

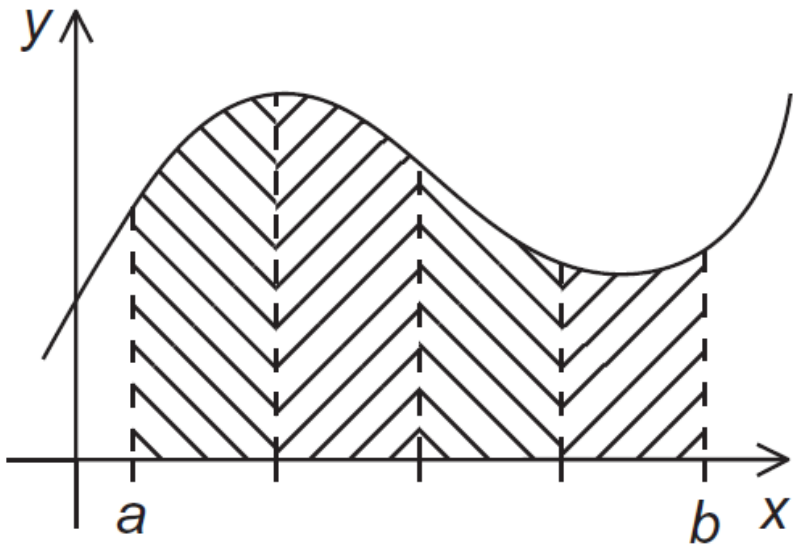


Trapezoidal rule in mpi
基于MPI的梯形积分法

The Trapezoidal Rule



(a)



(b)

The Trapezoidal Rule（梯形积分法）

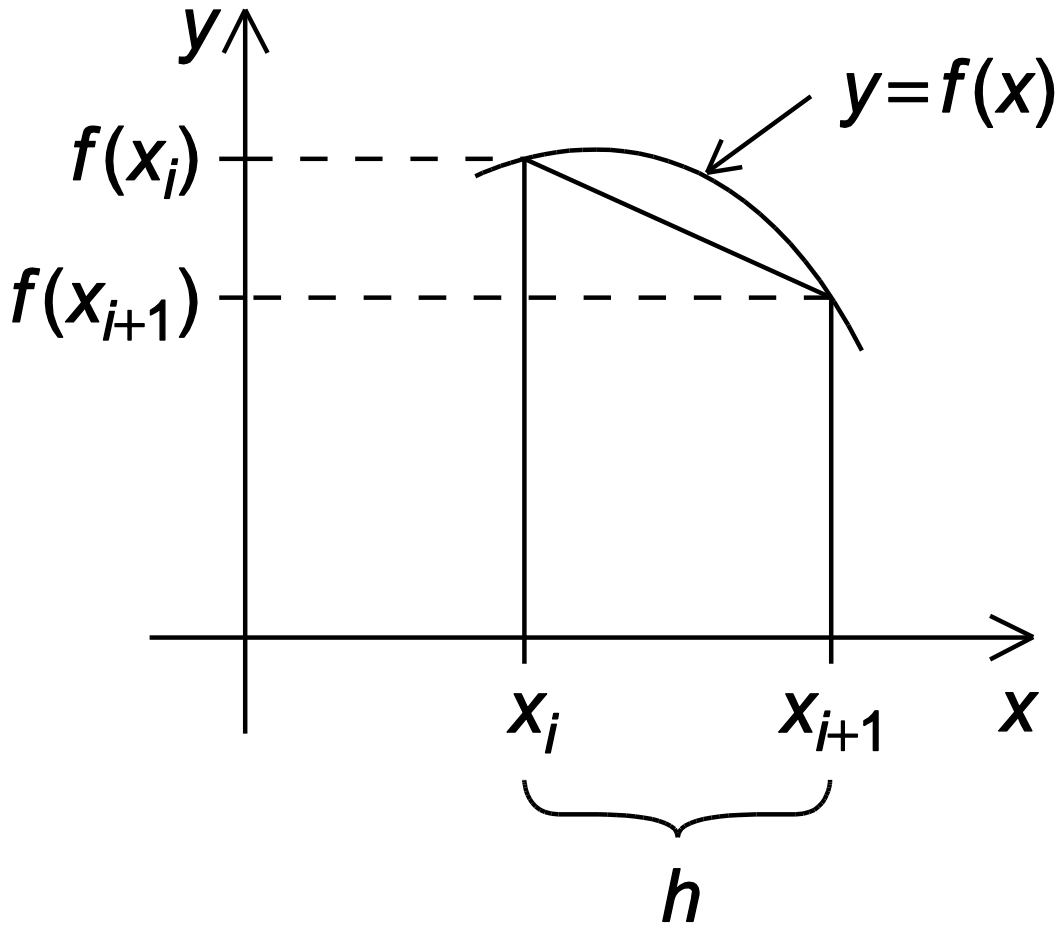
$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h,\ \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b$$

$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

One trapezoid



Pseudo-code for a serial program

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

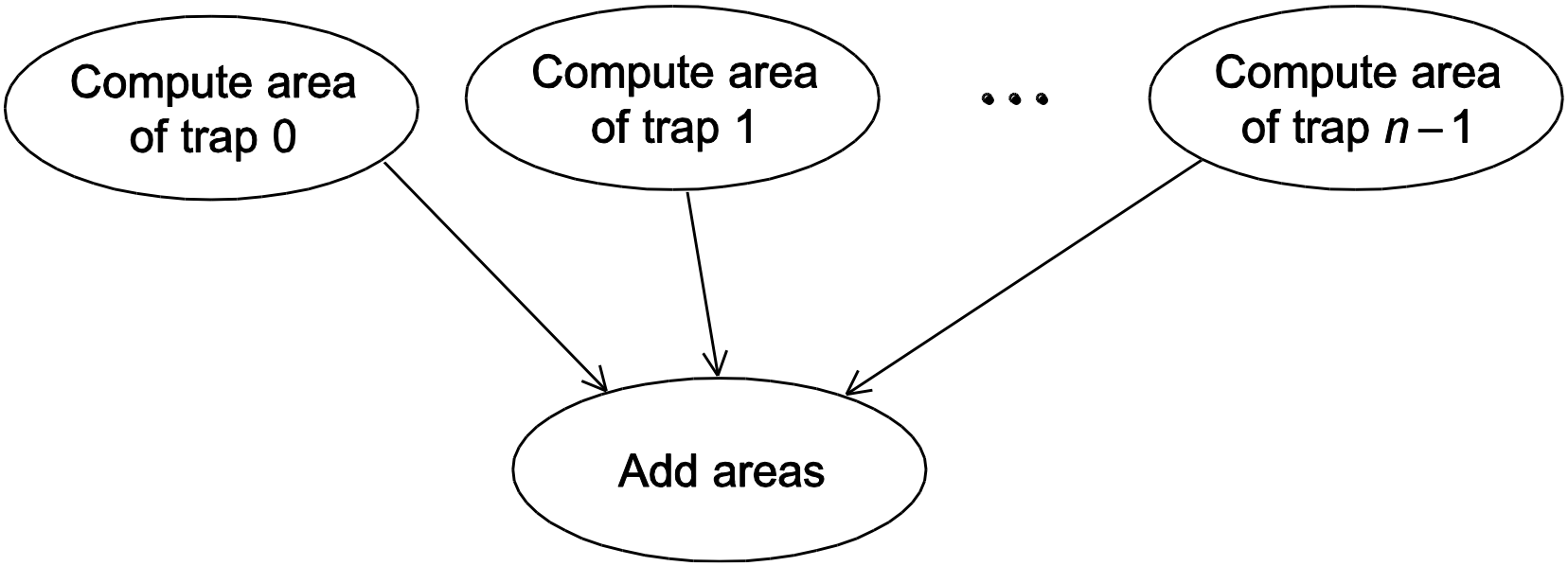
Parallelizing the Trapezoidal Rule

1. Partition problem solution into **tasks**.
2. Identify **communication channels** between tasks.
3. Aggregate tasks into **composite tasks**.
4. **Map** composite tasks to cores.

Parallel pseudo-code

```
1     Get a, b, n;
2     h = (b-a)/n;
3     local_n = n/comm_sz;
4     local_a = a + my_rank*local_n*h;
5     local_b = local_a + local_n*h;
6     local_integral = Trap(local_a, local_b, local_n, h);
7     if (my_rank != 0)
8        Send local_integral to process 0;
9     else /* my_rank == 0 */
10       total_integral = local_integral;
11       for (proc = 1; proc < comm_sz; proc++) {
12          Receive local_integral from proc;
13          total_integral += local_integral;
14       }
15    }
16    if (my_rank == 0)
17       print result;
```

Tasks and communications for Trapezoidal Rule



First version (1) 程序3-2

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;          /* h is the same for all processes */
12    local_n = n/comm_sz;  /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
```

First version (2)

```
21      } else {
22         total_int = local_int;
23         for (source = 1; source < comm_sz; source++) {
24            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            total_int += local_int;
27         }
28      }
29
30      if (my_rank == 0) {
31         printf("With n = %d trapezoids, our estimate\n", n);
32         printf("of the integral from %f to %f = %.15e\n",
33             a, b, total_int);
34      }
35      MPI_Finalize();
36      return 0;
37   } /*  main  */
```

First version (3)

```
1  double Trap(
2        double left_endpt  /* in */,
3        double right_endpt /* in */,
4        int    trap_count  /* in */,
5        double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1; i++) {
11       x = left_endpt + i*base_len;
12       estimate += f(x);
13    }
14    estimate = estimate*base_len;
15
16    return estimate;
17 } /*  Trap  */
```

Dealing with I/O

```
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
          my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

Each process just prints a message.

Running with 6 processes

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

unpredictable output

Input

- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.
- Process 0 must read the data (scanf) and send to the other processes.

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

Function for reading user input

```
void Get_input(
      int       my_rank    /* in  */,
      int       comm_sz    /* in  */,
      double*   a_p        /* out */,
      double*   b_p        /* out */,
      int*      n_p        /* out */) {
   int dest;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else { /* my_rank != 0 */
      MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
   }
}  /* Get_input */
```

Collective communication

Tree-structured communication

1. In the first phase:

   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.

   (b) Processes 0, 2, 4, and 6 add in the received values.

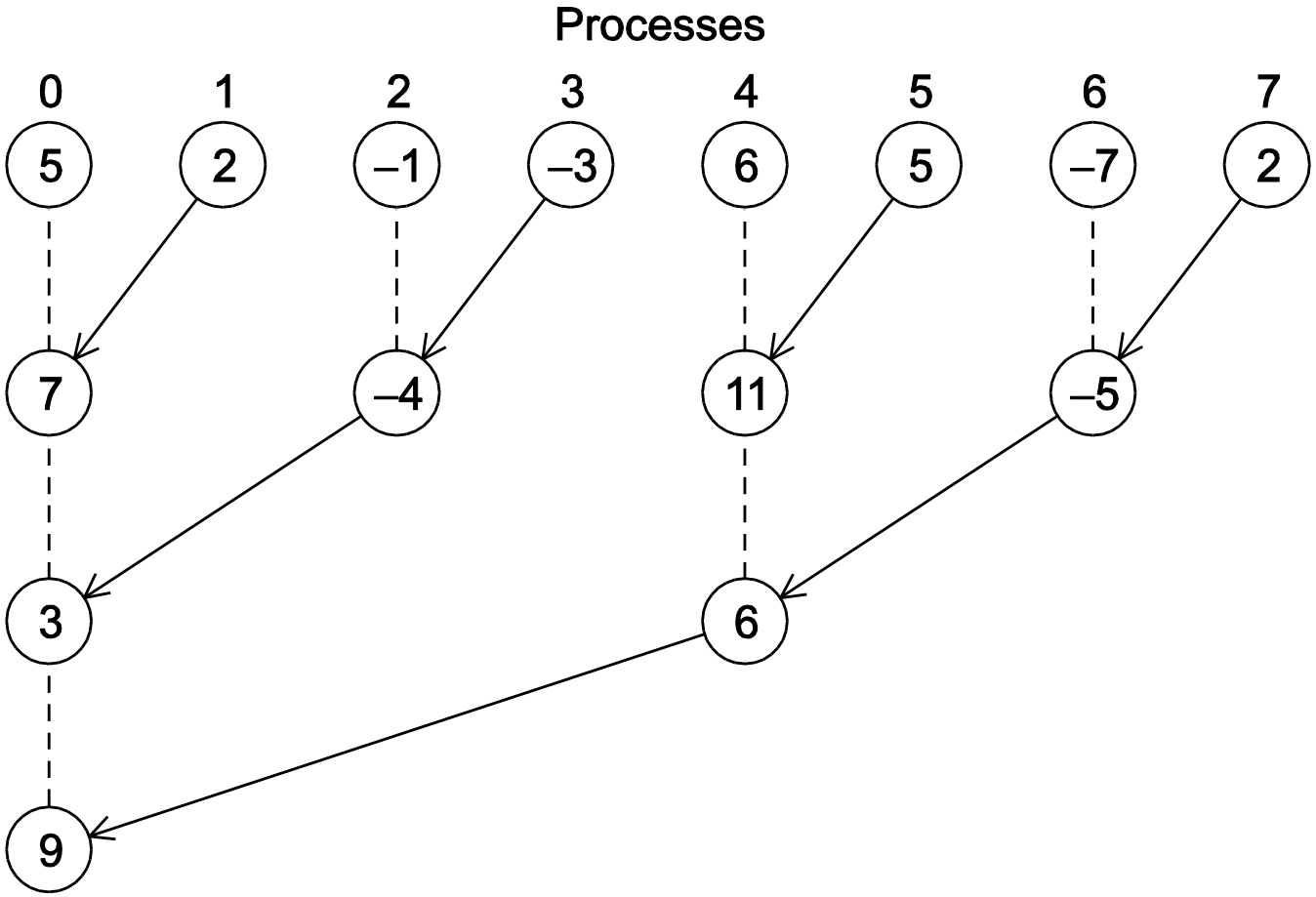
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.

   (d) Processes 0 and 4 add the received values into their new values.

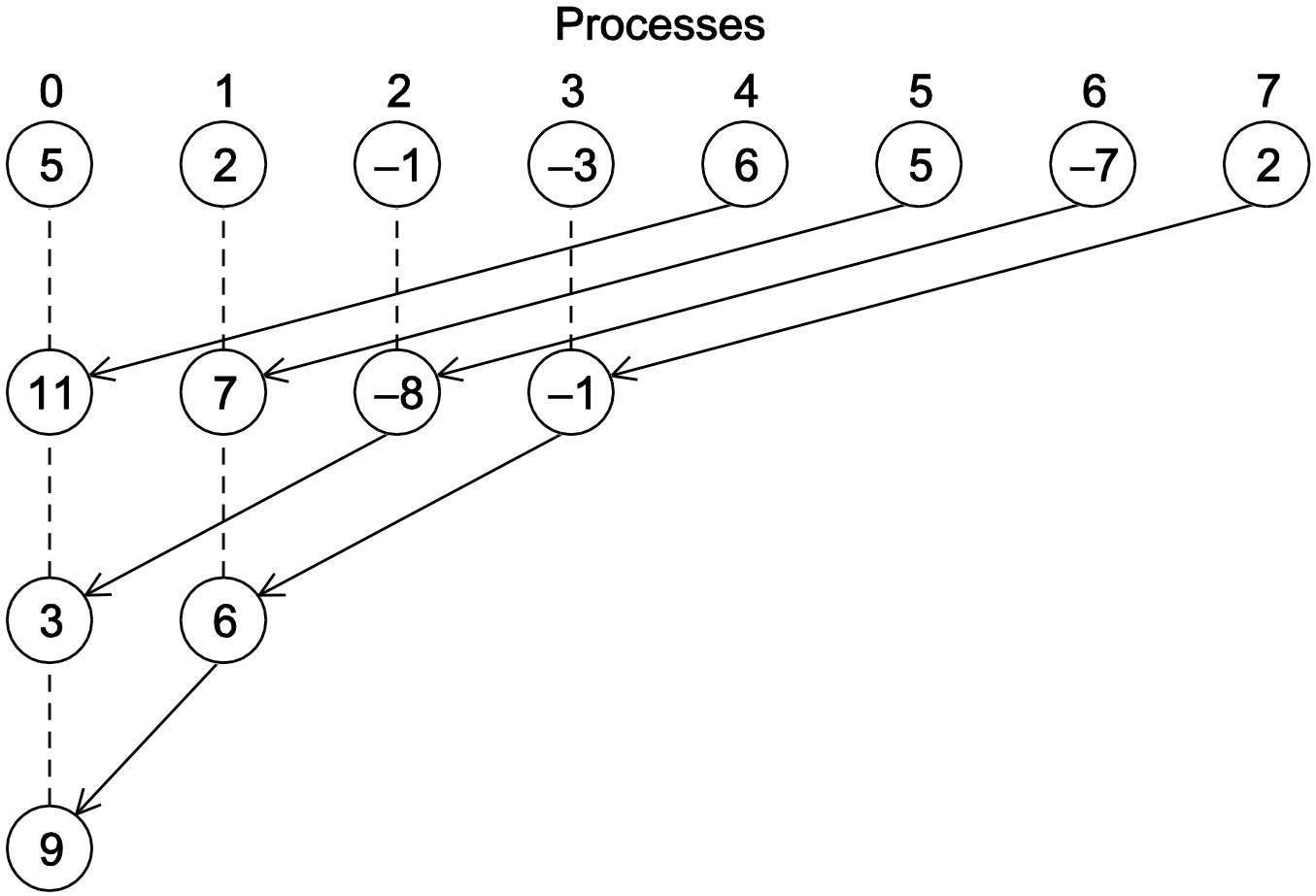
2. (a) Process 4 sends its newest value to process 0.

   (b) Process 0 adds the received value to its newest value.

A tree-structured global sum



An alternative tree-structured global sum



MPI_Reduce

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

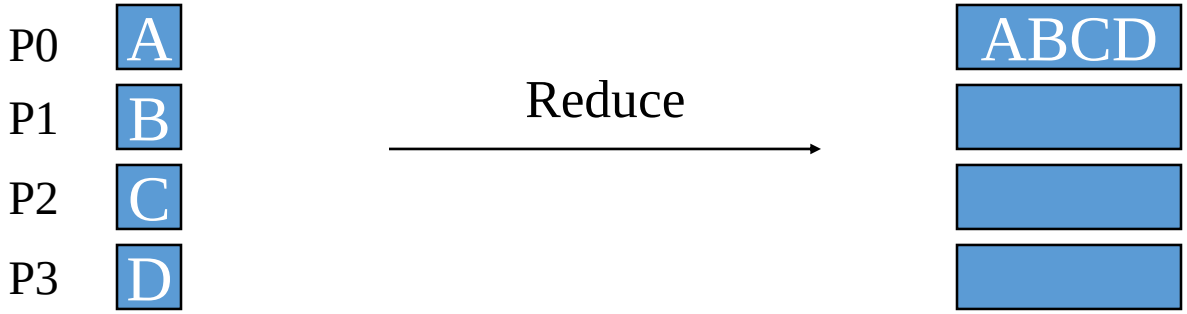
```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

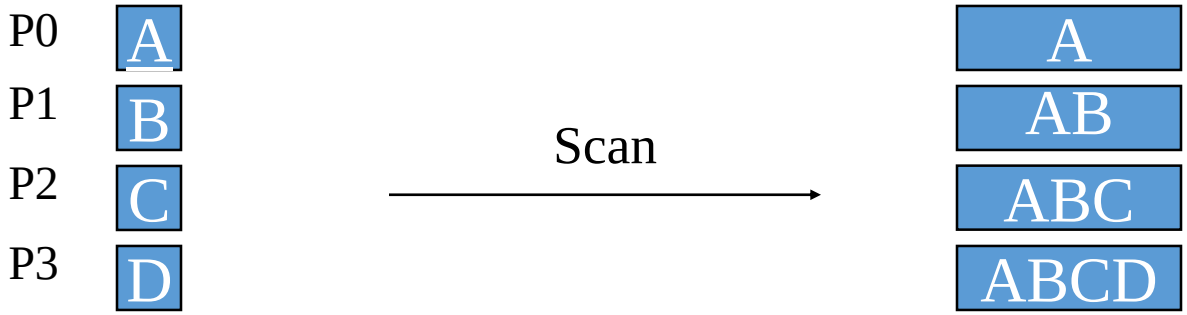
```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Predefined reduction operators in MPI

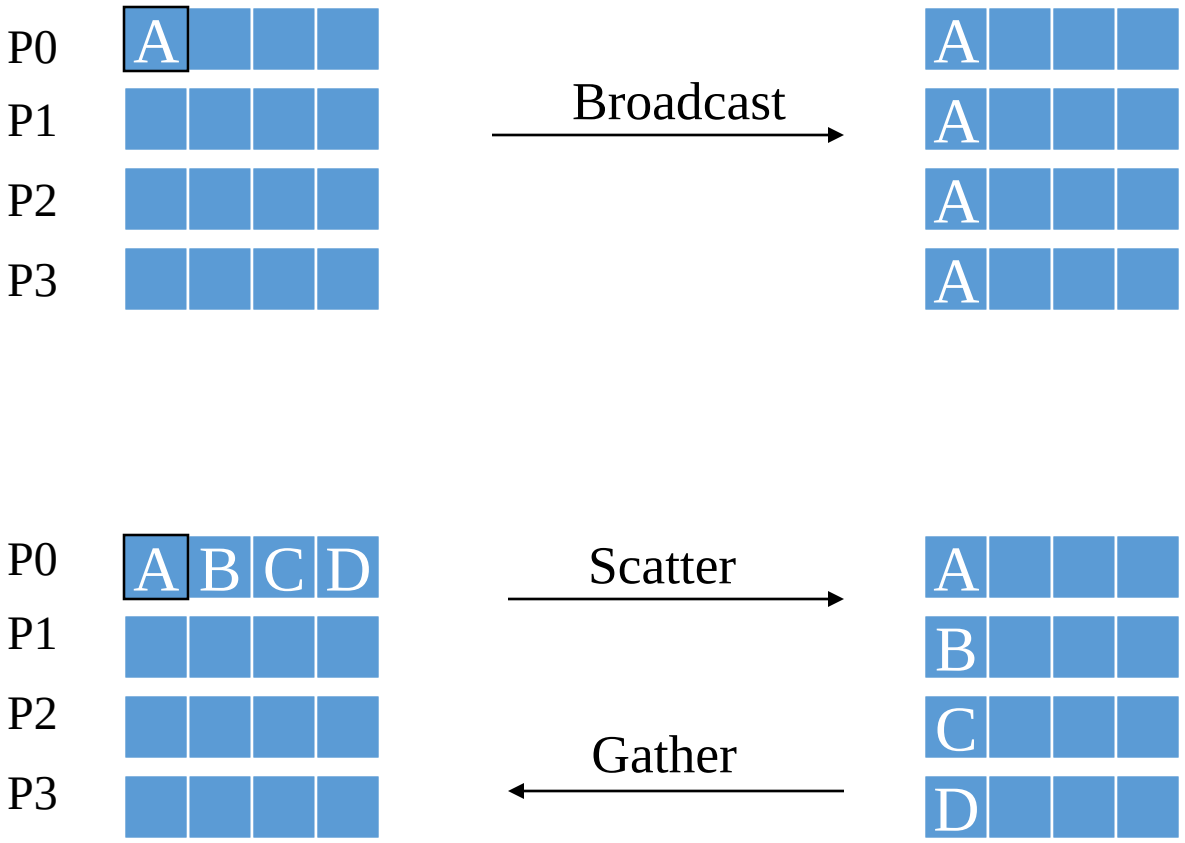
Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of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

Collective Data Movement

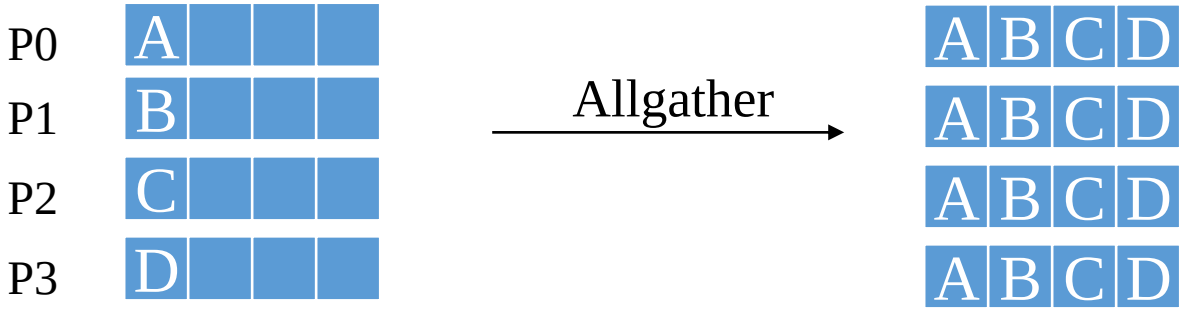




Collective Data Movement



Collective Data Movement



Collective vs. Point-to-Point Communications

- All the 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 dest_process 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 output_data_p argument is only used on dest_process.

- However, all of the processes still need to pass in an actual argument corresponding to output_data_p, 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 (调用顺序）**.

Example (1)

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, ...)`

b = 4, d = 5

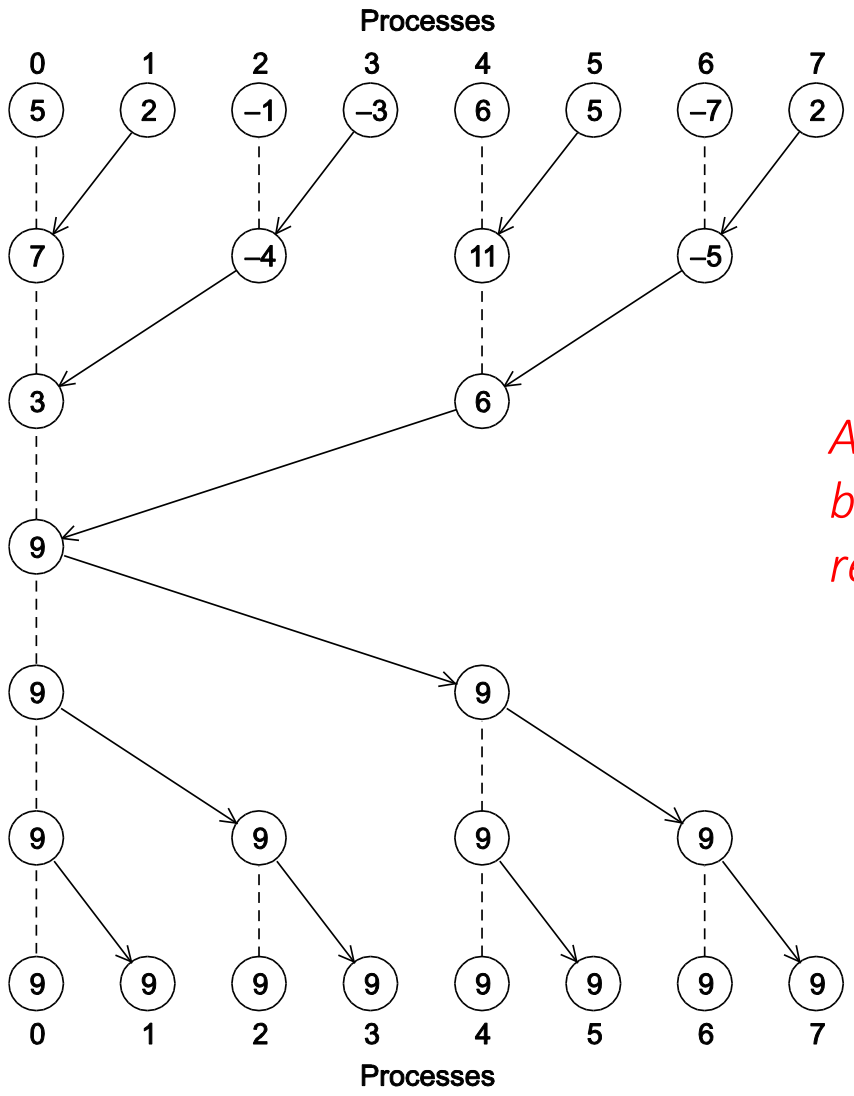
Multiple calls to MPI_Reduce,

Suppose that each process calls MPI_Reduce with operator MPI_SUM, and destination process 0.

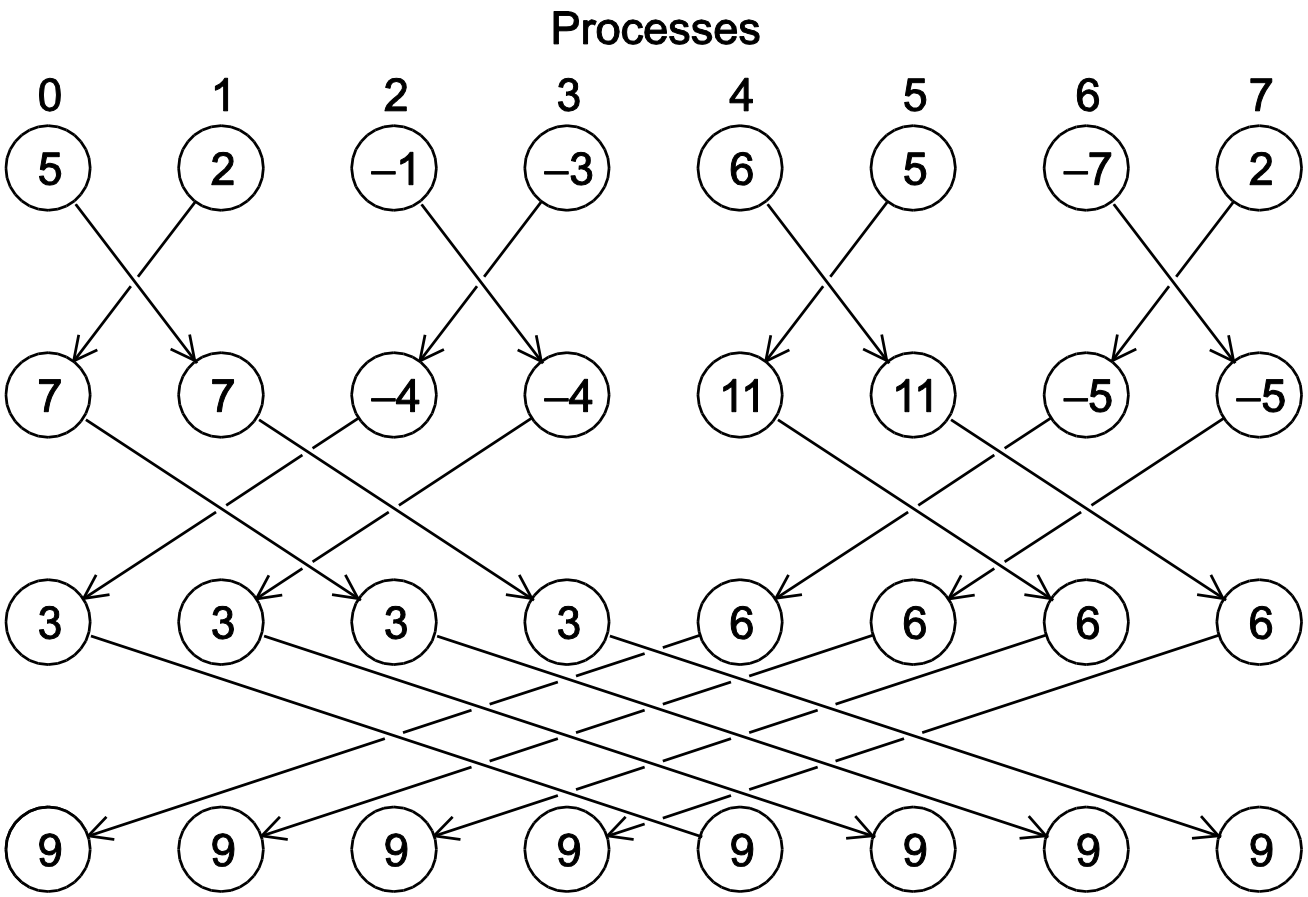
MPI_Allreduce

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```
int MPI_Allreduce(
          void*         input_data_p   /* in  */,
          void*         output_data_p  /* out */,
          int           count          /* in  */,
          MPI_Datatype  datatype       /* in  */,
          MPI_Op        operator       /* in  */,
          MPI_Comm      comm           /* in  */);
```



A global sum followed by distribution of the result.



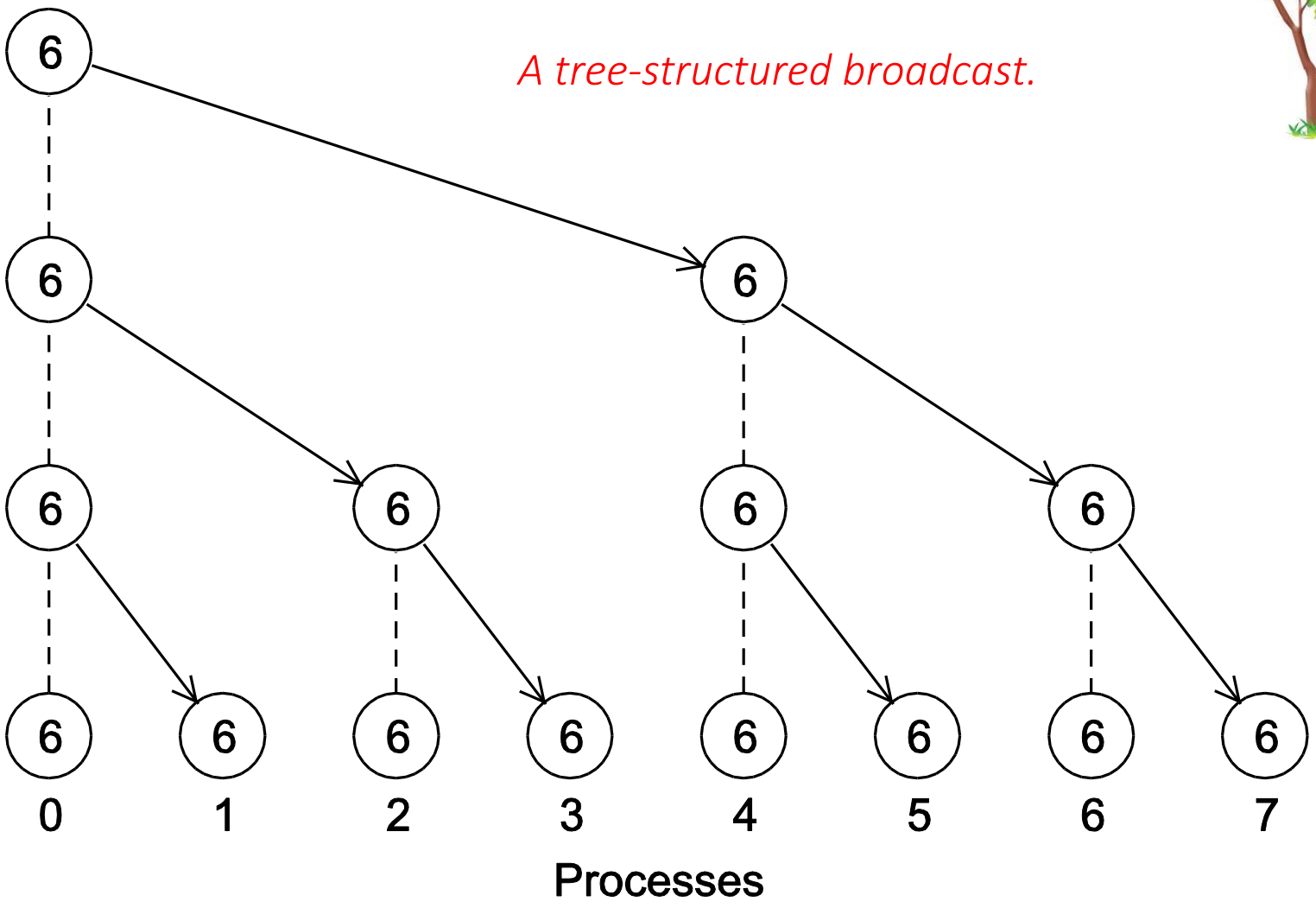
A butterfly-structured global sum.

Broadcast

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

```
int MPI_Bcast(
        void*           data_p          /* in/out */,
        int             count           /* in     */,
        MPI_Datatype    datatype        /* in     */,
        int             source_proc     /* in     */,
        MPI_Comm        comm            /* in     */);
```

A tree-structured broadcast.



A version of Get_input that uses MPI_Bcast

```
void Get_input(
      int       my_rank  /* in  */,
      int       comm_sz  /* in  */,
      double*   a_p      /* out */,
      double*   b_p      /* out */,
      int*      n_p      /* out */) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

Data distributions

$$
\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}
$$

Compute a vector sum.

Serial implementation of vector addition

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Different partitions of a 12-component vector among 3 processes

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Partitioning options

- Block partitioning
  - Assign blocks of consecutive components to each process.
- Cyclic partitioning
  - Assign components in a round robin fashion.
- Block-cyclic partitioning
  - Use a cyclic distribution of blocks of components.

Parallel implementation of vector addition

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

伪代码

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

Reading and distributing a vector

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```
int MPI_Gather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            dest_proc   /* in  */,
      MPI_Comm       comm        /* in  */);
```

Print a distributed vector (1)

```
void Print_vector(
      double    local_b[]  /* in */,
      int       local_n    /* in */,
      int       n          /* in */,
      char      title[]    /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {

   double* b = NULL;
   int i;
```

Print a distributed vector (2)

```
   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Print_vector */
```

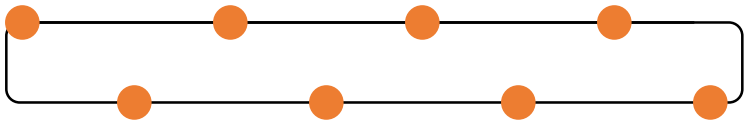
Allgather

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- As usual, recv_count is the amount of data being received from each process.

```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

AllGather

Ring Algorithm



$T_{ring} = \alpha (p-1) + \beta n (p-1)/p$

α： latency

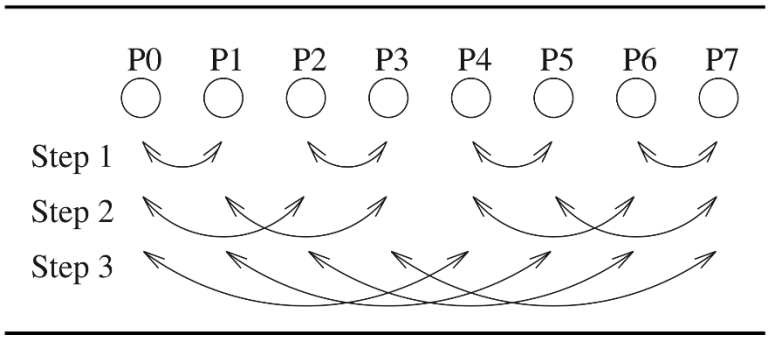
β： time to transmit a byte

n: number of bytes of all processes

p: number of processes

- **At time t**: **send** the data you received at time t-1 **to** your **right**, and **receive** new data **from** your **left**.
- **At time 0**, send your original data
- Optimal bandwidth, high latency

Recursive Doubling Algorithm



$T_{rec\text{-}dbl} = \alpha \lg(p) + \beta n (p-1)/p$

- **At time t**: process i exchanges (send/recv) all your current data (its original data plus anything received so far) with process $i \pm 2^t$
- Data exchanged at each step: n/p, 2n/p, 4n/p,…, $2^{\lg(p)-1}$ n/p
- Tricky for non-power-of-two

AllGather Performance

- Similar ideas are used in other collectives (e.g. **recursive halving** instead of recursive doubling for **reduce-scatter**) with different local computations (e.g. for **allreduce**, perform a **local reduction** at each step instead of **concatenating** data as in allgather)

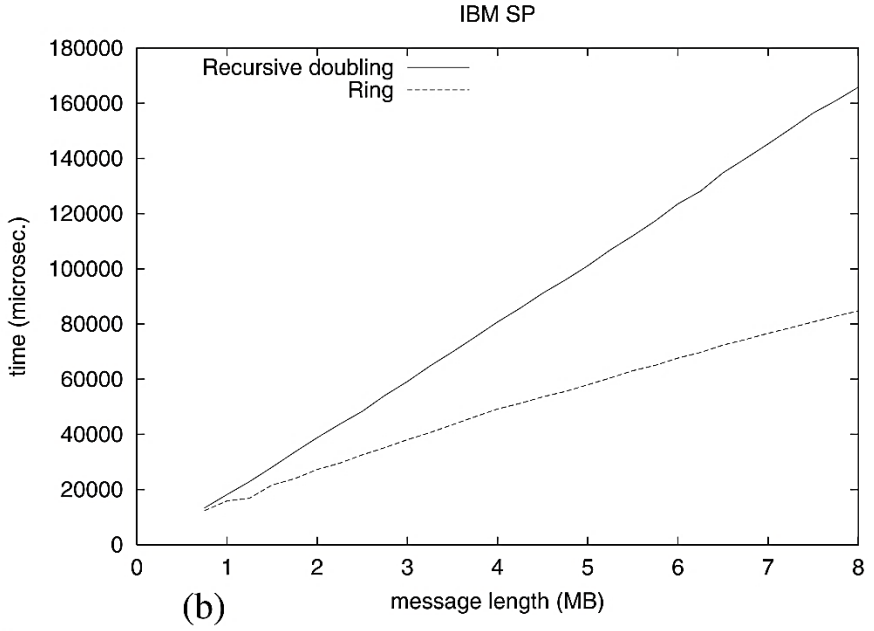


Fig. 5 Ring algorithm versus recursive doubling for long-message allgather (64 nodes). The size on the x-axis is the total amount of data gathered on each process.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH." *The International Journal of High Performance Computing Applications* 19.1 (2005): 49-66

MPI Derived Datatype

Derived datatypes

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- The idea is that if a function that sends data knows this 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.

Derived datatypes

- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

Variable	Address
a	24
b	40
n	48

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

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 */);
```

MPI_Get_address

- Returns the address of the memory location referenced by location_p.
- The special type MPI_Aint is an integer type that is big enough to store an address on the system.

```
int MPI_Get_address(
      void*      location_p   /* in  */,
      MPI_Aint*  address_p    /* out */);
```

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_mpi_t_p  /* in/out */);
```

MPI_Type_free

- When we're finished with our new type, this frees any additional storage used.

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

Get input function with a derived datatype (1)

```
void Build_mpi_type(
      double*         a_p              /* in  */,
      double*         b_p              /* in  */,
      int*            n_p              /* in  */,
      MPI_Datatype*   input_mpi_t_p    /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};
```

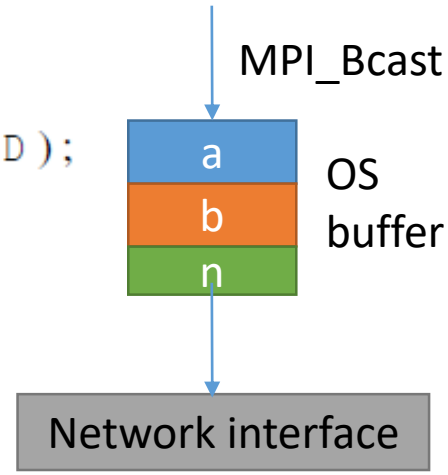
Get input function with a derived datatype (2)

```
    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);
    array_of_displacements[1] = b_addr-a_addr;
    array_of_displacements[2] = n_addr-a_addr;
    MPI_Type_create_struct(3, array_of_blocklengths,
          array_of_displacements, array_of_types,
          input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
}   /* Build_mpi_type */
```

Get input function with a derived datatype (3)

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

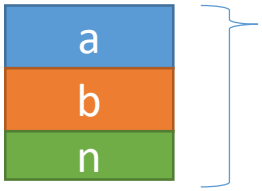
Variable	Address
a	24
b	40
n	48



Difference with C Struct

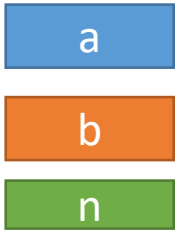
```
struct range_integral_t {
  double a;
  double b;
  int n;
};
```

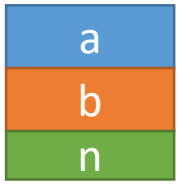
new or malloc



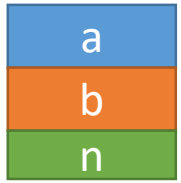
struct range_integral_t* range_p

```
int main () {
  double a;
……
  double b;
  int n;
};
```

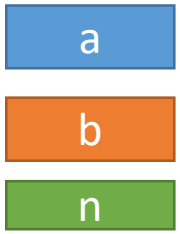




Buffer



Buffer




```
int main () {
  double a;
……
  double b;
  int n;
};
```

MPI_Type_create_struct

Send by Proc 0

MPI_Type_create_struct

Recv by Proc 1

Other derived datatype

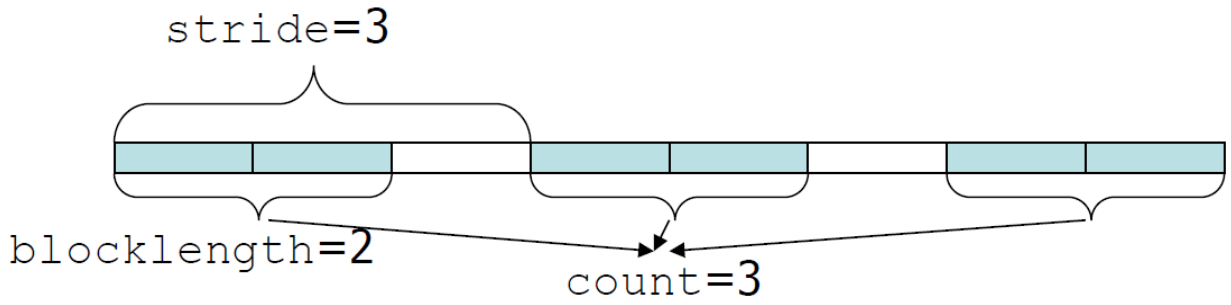
- MPI_Type_contiguous (int count, MPI_Datatype datatype, MPI_Datatype *newtype);
  - **count** elements of the same datatype forming a contiguous chunk in the memory
- Input datatype can be a derived datatype
  - End of one element of the derived datatype has to be exactly at the beginning of the next element of the derived datatype

```
int myvec[4];
MPI_Type_contiguous ( 4, MPI_INT, &mybrandnewdatatype);
MPI_Type_commit ( &mybrandnewdatatype );
MPI_Send ( myvec, 1, mybrandnewdatatype, … );
```

Other derived datatype

- MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype datatype, MPI_Datatype *newtype);
  - count blocks of blocklength elements of the same datatype
  - Between the start of each block there are stride elements of the same datatype

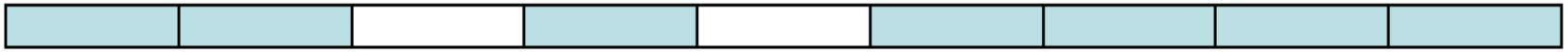
```
double myvec[8];
dtype = MPI_DOUBLE;  stride = 3;
blength = 2;  count = 3;
MPI_Type_vector (count,blength,stride,dtype,&newtype);
MPI_Type_commit (&newtype);
MPI_Send ( myvec, 1, newtype, … );
```



Other derived datatype

- MPI_Type_indexed(int count, int blocklengths[], int displacements[], MPI_Datatype datatype, MPI_Datatype *newtype);
  - The number of elements per block do not have to be identical
  - *displacements* gives the distance from the 'base' to the beginning of the block in multiples of the used datatype

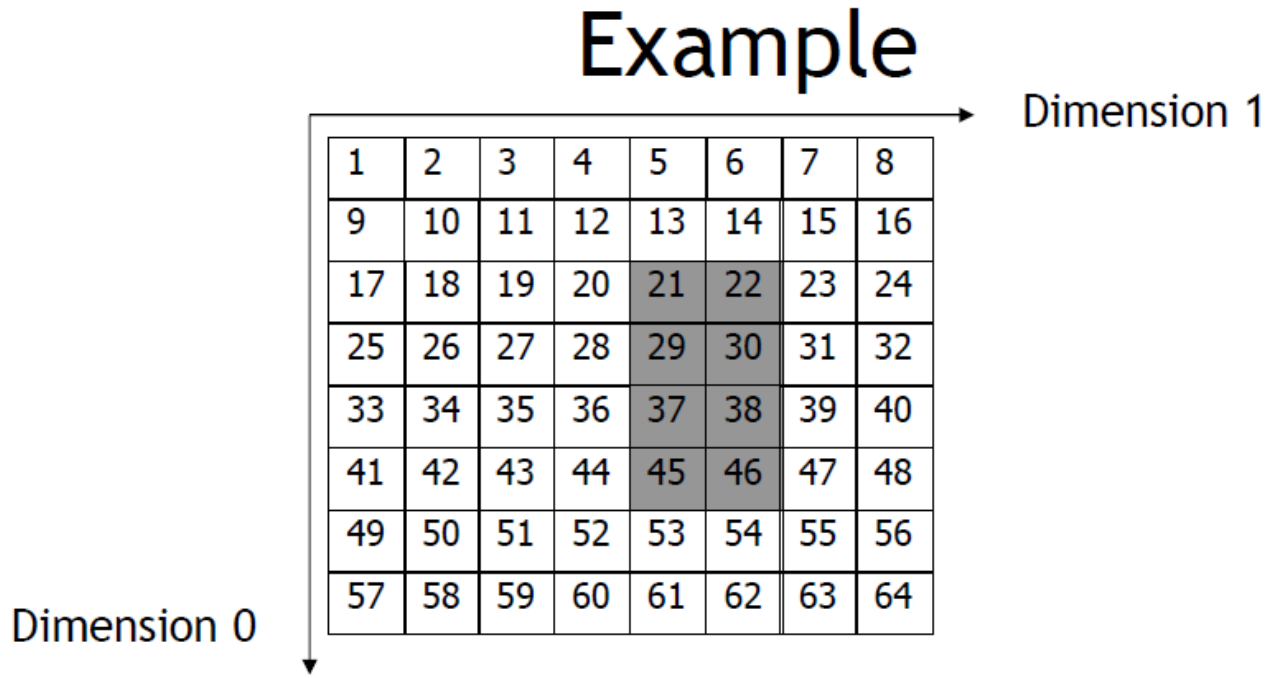
count = 3
blocklengths[0] = 2 displacements[0] = 0
blocklengths[1] = 1 displacements[1] = 3
blocklengths[2] = 4 displacements[2] = 5



Other derived datatype

- MPI_Type_create_subarray (int ndims, int sizes[], int subsizes[], int starts[], int order, MPI_Datatype datatype,MPI_Datatype *newtype);
  - Define sub-matrices of n-dimensional data
  - sizes[]: dimension of the entire matrix
  - subsizes[]: dimensions of the submatrix described by the derived data type
  - starts[]: array describing the beginning of the submatrices
  - Order: MPI_ORDER_C for row-major order or MPI_ORDER_FORTRAN for column-major data

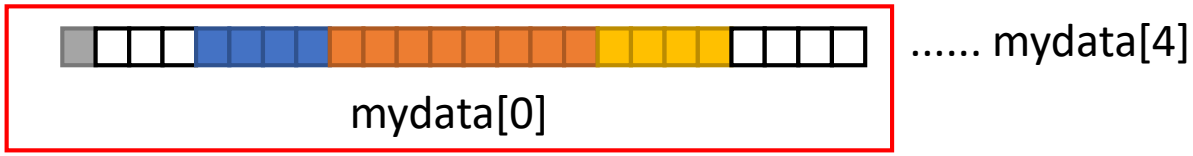
Other derived datatype




```
ndims = 2;
sizes[0] = 8; sizes[1] = 8;
subsizes[0]= 4; subsizes[1] = 2
starts[0] = 2; starts[1] = 4;
MPI_Type_create_subarray ( ndims, sizes, subsizes, starts, MPI_ORDER_C,
                                        MPI_DOUBLE, &newtype);
```

Problem of derived datatype

```
typedef struct {
    char a;
    int b;
    double c;
    float d;
} mystruct;
mystruct mydata[5];
```



- If we just want to send elements **b** and **c** of the structure?

Problem of derived datatype

MPI_Get_address (&(mydata[0], &baseaddr);
MPI_Get_address (&(mydata[0].b, &addr1);
MPI_Get_address (&(mydata[0].c, &addr2);
displ[0] = addr1 - baseaddr;
displ[1] = addr2 – baseaddr;
dtype[0] = MPI_INT; blength[0] = 1;
dtype[1] = MPI_DOUBLE; blength[1] = 1;
MPI_Type_create_struct (2,blength,displ,dtype,&newtype);
MPI_Type_commit (&newtype);
MPI_Send (mydata, 1, newtype,…); //it is ok if we send one struct instance.
MPI_Send (mydata, 5, newtype, …); //If send more instances, received data will be wrong

- Memory layout

- What we send is

..... mydata[4]

- What we wanted to send is

Problem of derived datatype

- Solution to the problem
- MPI_Type_create_resized (MPI_Datatype datatype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype);
  - Reset the low-bound and up-bound marker of a derived datatype

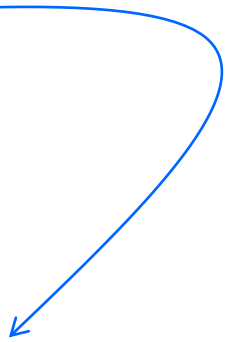
```
MPI_Get_address ( &(mydata[0]), &baseaddr);
MPI_Get_address ( &(mydata[0].b), &addr1);
MPI_Get_address ( &(mydata[0].c), &addr2);
MPI_Get_address ( &(mydata[1]), &addr3);
displ[0] = addr1 - baseaddr;
displ[1] = addr2 – baseaddr;
extent = addr3 – baseaddr;
dtype[0] = MPI_INT;            blength[0] = 1;
dtype[1] = MPI_DOUBLE;      blength[1] = 1;
MPI_Type_create_struct (4,blength,displ,dtype,&newtype);
MPI_Type_create_resized ( newtype, baseaddr, extent, &newtype2);
MPI_Type_commit ( &newtype2);
```

Performance evaluation

Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);
```



```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

Average/least/most execution time spent by individual process

```
int myrank, numprocs;
double mytime, maxtime, mintime, avgtime; /*variables used for gathering timing statistics*/

MPI_Comm_rank(MPI_COMM_WORLD, &myrank); MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Barrier(MPI_COMM_WORLD); /*synchronize all processes*/

mytime = MPI_Wtime(); /*get time just before work section */
work();
mytime = MPI_Wtime() - mytime; /*get time just after work section*/ /*compute max, min, and average timing statistics*/

MPI_Reduce(&mytime, &maxtime, 1, MPI_DOUBLE,MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&mytime, &mintime, 1, MPI_DOUBLE, MPI_MIN, 0,MPI_COMM_WORLD);
MPI_Reduce(&mytime, &avgtime, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

if (myrank == 0)  {
        avgtime /= numprocs;
        printf("Min: %lf Max: %lf Avg: %lf\n", mintime, maxtime,avgtime);
 }
```

Elapsed serial time

- In this case, you don't need to link in the MPI libraries.
- Returns time in microseconds elapsed from some point in the past.

```
#include "timer.h"
. . .
double now;
. . .
GET_TIME(now);
```

Elapsed serial time

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.
- Blocks until all processes in the group of the communicator comm call it.

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

MPI_Barrier

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Speedup

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}$$

Efficiency

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}$$

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Scalability

- A program is scalable if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of pro

Scalability

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable.

- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be weakly scalable.

A parallel sorting algorithm

Sorting

- n keys and p = number of processes.
- n/p keys assigned to each process.
- No restrictions on which keys are assigned to which processes.
- When the algorithm terminates:
  - The keys assigned to each process should be sorted in (say) increasing order.
  - If $0 \leq q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.

Serial bubble sort

```
void Bubble_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2; list_length--)
        for (i = 0; i < list_length-1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }

}  /* Bubble_sort */
```

Odd-even transposition sort

- A sequence of phases.
- Even phases, compare swaps:

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots$$

- Odd phases, compare swaps:

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots.$$

Example

Start: 5, 9, 4, 3

Even phase: compare-swap (5,9) and (4,3)

getting the list 5, 9, 3, 4

Odd phase: compare-swap (9,3)

getting the list 5, 3, 9, 4

Even phase: compare-swap (5,3) and (9,4)

getting the list 3, 5, 4, 9

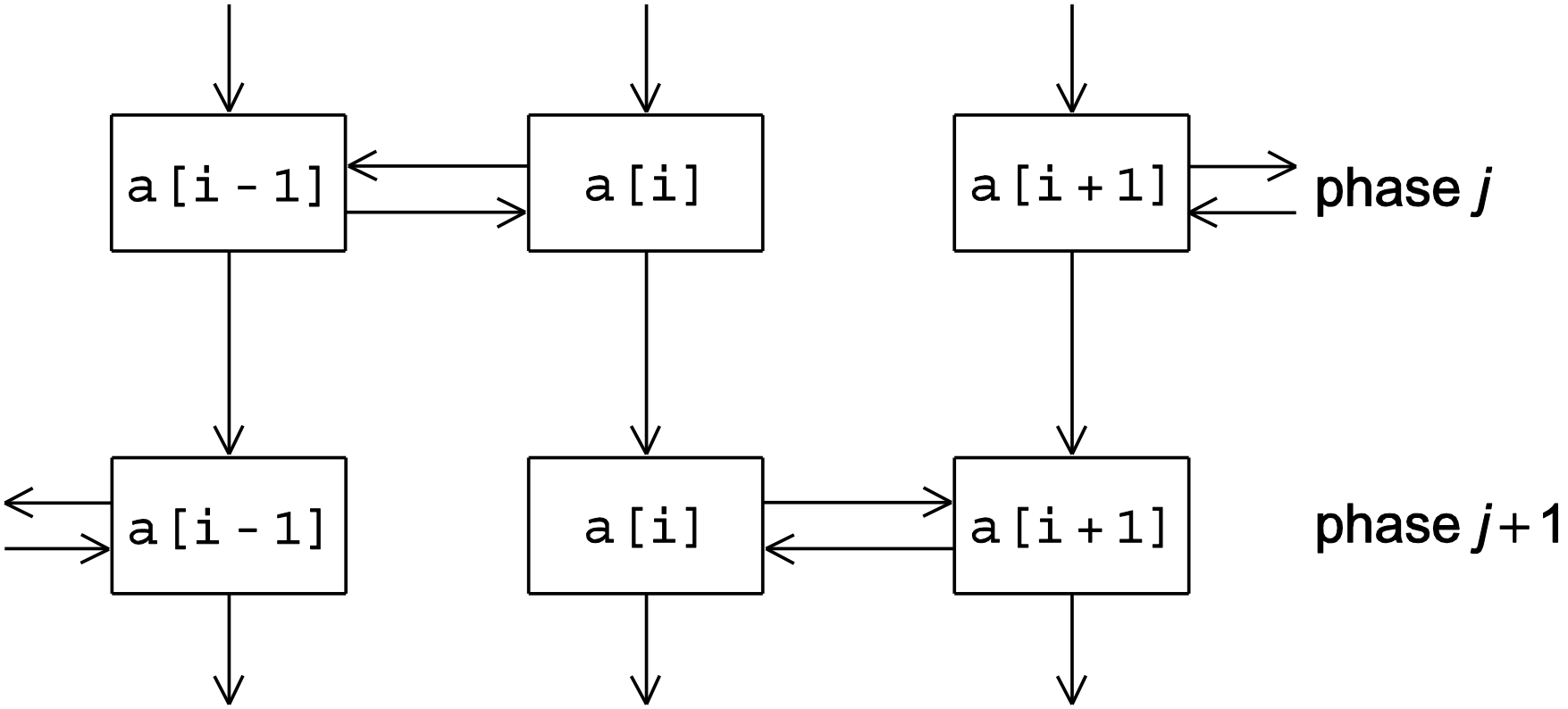
Odd phase: compare-swap (5,4)

getting the list 3, 4, 5, 9

Serial odd-even transposition sort

```
void Odd_even_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
}  /* Odd_even_sort */
```

Communications among tasks in odd-even sort



Tasks determining a[i] are labeled with a[i].

Parallel odd-even transposition sort

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Pseudo-code

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Compute_partner

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)       /* Odd rank */
      partner = my_rank - 1;
   else                        /* Even rank */
      partner = my_rank + 1;
else                         /* Odd phase */
   if (my_rank % 2 != 0)       /* Odd rank */
      partner = my_rank + 1;
   else                        /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

Safety in MPI programs

- The MPI standard allows MPI_Send to behave in two different ways:
  - it can simply copy the message into an MPI managed buffer and return,
  - or it can block until the matching call to MPI_Recv starts.

Safety in MPI programs

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking.
- Relatively small messages will be buffered by MPI_Send.
- Larger messages, will cause it to block.

Safety in MPI programs

- If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock.

- Each process is blocked waiting for an event that will never happen.

Safety in MPI programs

- A program that relies on MPI provided buffering is said to be unsafe.

- Such a program may run without problems for various sets of input, but it may hang or crash with other sets.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called "unsafe" because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	`Bsend(0)`
`Recv(1)`	`Recv(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

MPI_Ssend

- An alternative to MPI_Send defined by the MPI standard.
- The extra “s” stands for synchronous and MPI_Ssend is guaranteed to block until the matching receive starts.

```
int MPI_Ssend(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

Restructuring communication

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
      0, comm, MPI_STATUS_IGNORE.
```

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
} else {
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```

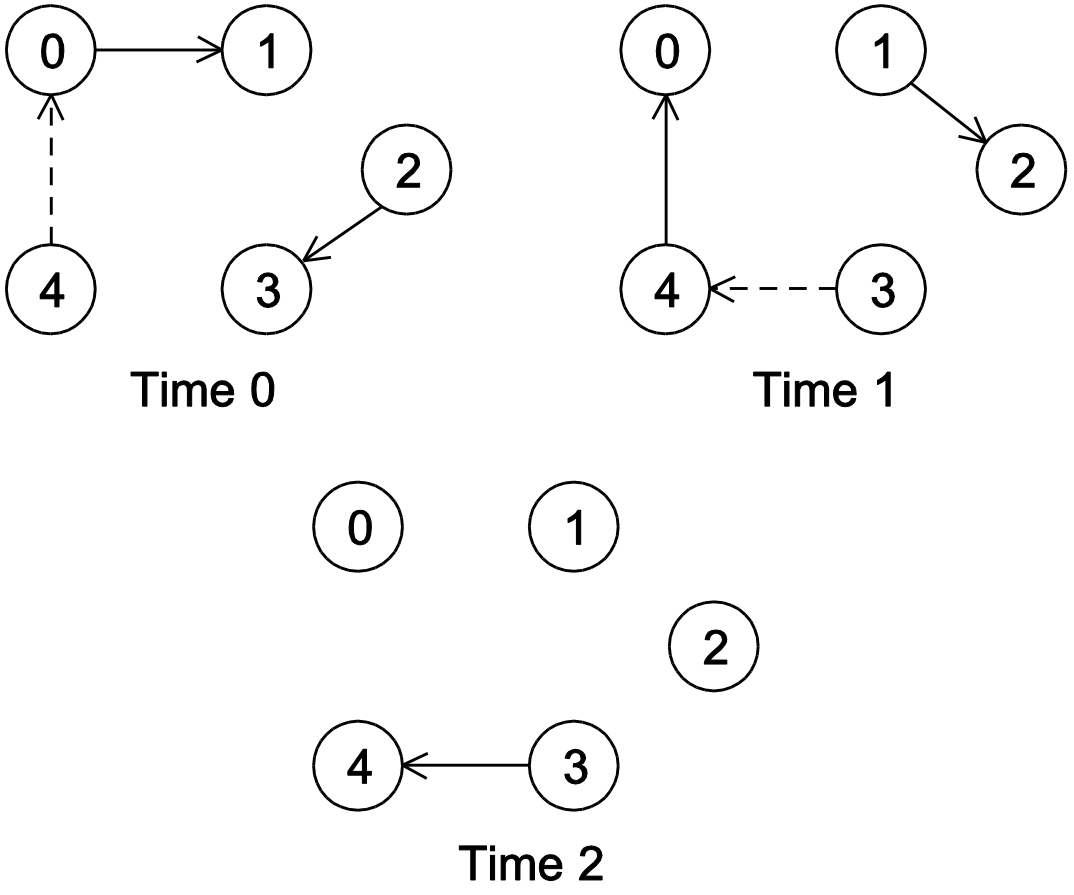
MPI_Sendrecv

- An alternative to scheduling the communications ourselves.
- Carries out a blocking send and a receive in a single call.
- The dest and the source can be the same or different.
- Especially useful because MPI schedules the communications so that the program won't hang or crash.

MPI_Sendrecv

```
int MPI_Sendrecv(
        void*           send_buf_p      /* in  */,
        int             send_buf_size   /* in  */,
        MPI_Datatype    send_buf_type   /* in  */,
        int             dest            /* in  */,
        int             send_tag        /* in  */,
        void*           recv_buf_p      /* out */,
        int             recv_buf_size   /* in  */,
        MPI_Datatype    recv_buf_type   /* in  */,
        int             source          /* in  */,
        int             recv_tag        /* in  */,
        MPI_Comm        communicator    /* in  */,
        MPI_Status*     status_p        /* in  */);
```

Safe communication with five processes



Parallel odd-even transposition sort

```
void Merge_low(
        int  my_keys[],      /* in/out     */
        int  recv_keys[],    /* in         */
        int  temp_keys[],    /* scratch    */
        int  local_n         /* = n/p, in  */) {
    int m_i, r_i, t_i;

    m_i = r_i = t_i = 0;
    while (t_i < local_n) {
        if (my_keys[m_i] <= recv_keys[r_i]) {
            temp_keys[t_i] = my_keys[m_i];
            t_i++; m_i++;
        } else {
            temp_keys[t_i] = recv_keys[r_i];
            t_i++; r_i++;
        }
    }

    for (m_i = 0; m_i < local_n; m_i++)
        my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

Run-times of parallel odd-even sort

Processes	Number of Keys (in thousands)				
	200	400	800	1600	3200
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

(times are in milliseconds)

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI_Request request;

MPI_Status status;

MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);

MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);

MPI_Wait(&request, &status);
```

(each request must be Waited on)

- One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

- Accessing the data buffer without waiting is undefined

MPI's Non-blocking Operations

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
  array_of_statuses)

MPI_Waitany(count, array_of_requests,
  &index, &status)

MPI_Waitsome(count, array_of_requests,
  array_of indices, array_of_statuses)
```

- There are corresponding versions of **`test`** for each of these.

Concluding Remarks (1)

- MPI or the Message-Passing Interface is a library of functions that can be called from C, C++, or Fortran programs.
- A communicator is a collection of processes that can send messages to each other.
- Many parallel programs use the single-program multiple data or SPMD approach.

Concluding Remarks (2)

- Most serial programs are deterministic: if we run the same program with the same input we'll get the same output.
- Parallel programs often don't possess this property.
- Collective communications involve all the processes in a communicator.

Concluding Remarks (3)

- When we time parallel programs, we're usually interested in elapsed time or "wall clock time".
- Speedup is the ratio of the serial run-time to the parallel run-time.
- Efficiency is the speedup divided by the number of parallel processes.

Concluding Remarks (4)

- If it's possible to increase the problem size (n) so that the efficiency doesn't decrease as p is increased, a parallel program is said to be scalable.
- An MPI program is unsafe if its correct behavior depends on the fact that MPI_Send is buffering its input.