

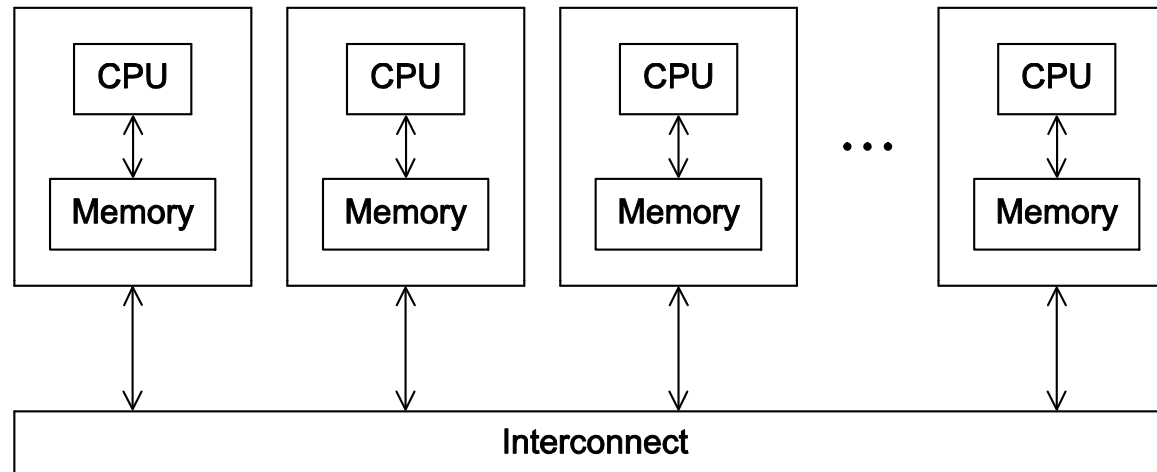
Chapter 3

Distributed Memory Programming with MPI

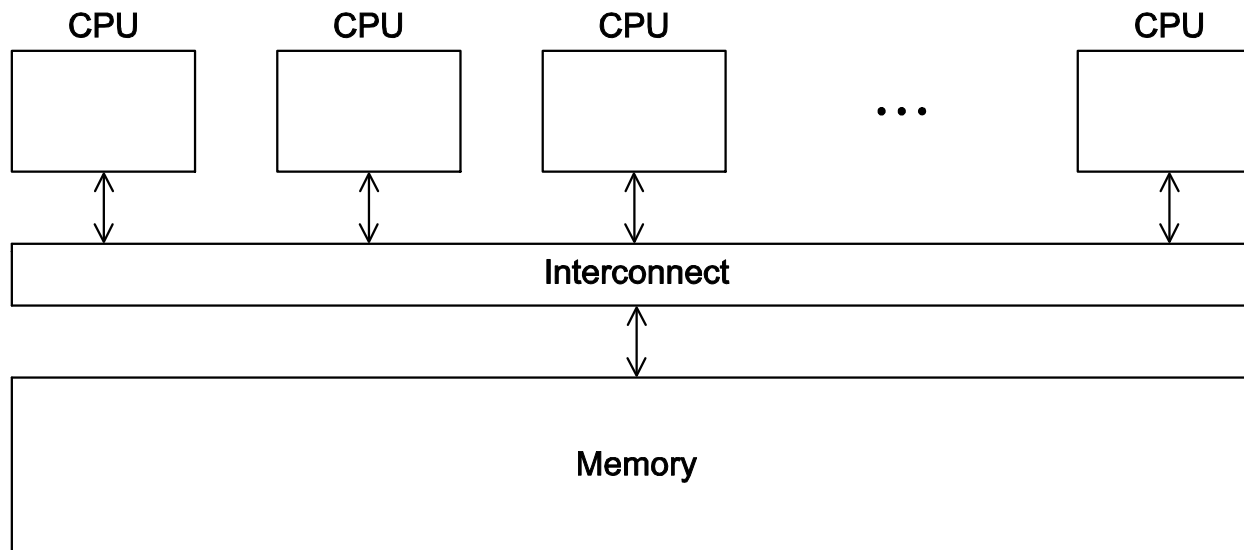
Roadmap

- 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)

Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.
- p processes are numbered $0, 1, 2, \dots, p-1$

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 */
```

Compilation

wrapper script to compile

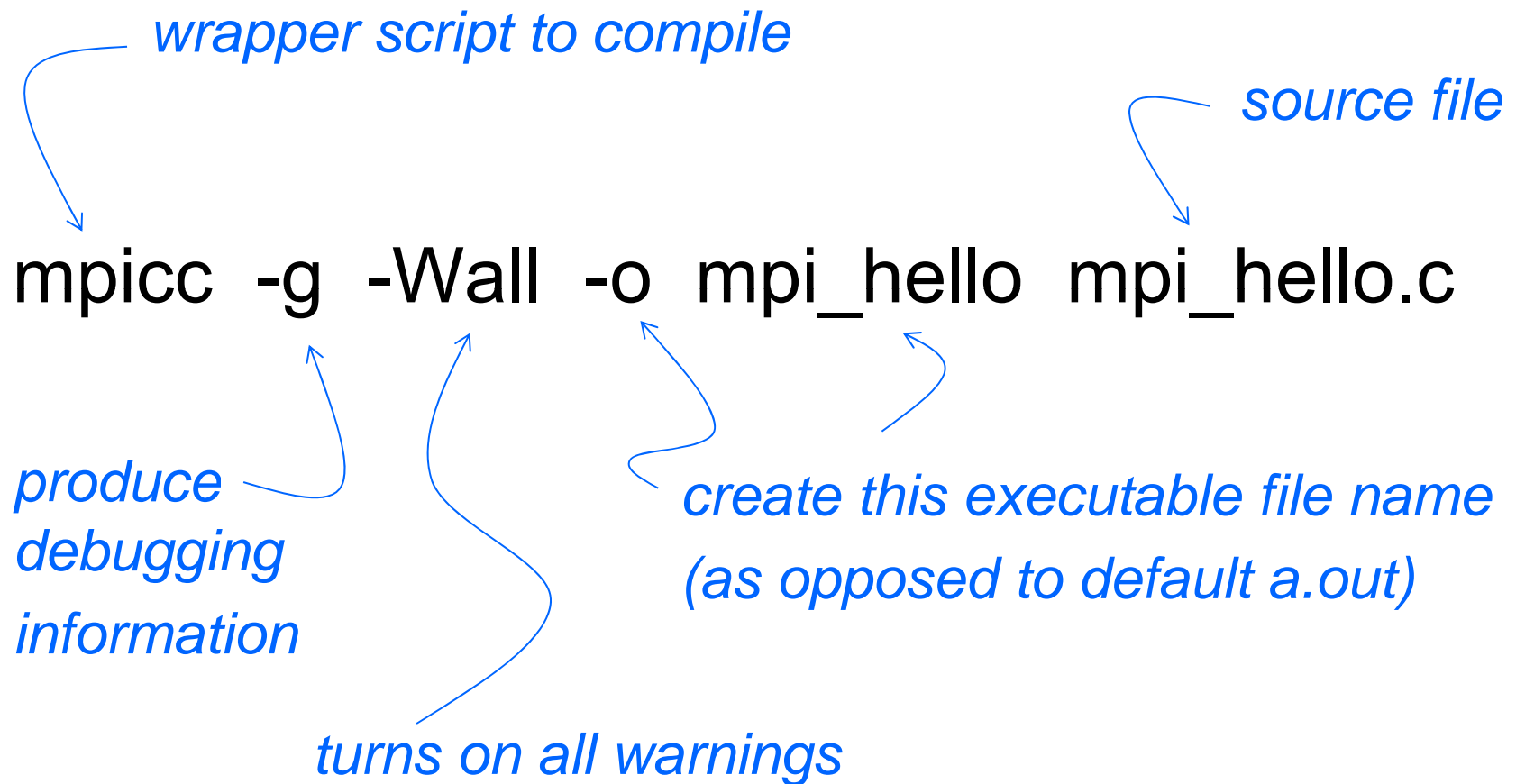
source file

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

produce debugging information

*create this executable file name
(as opposed to default a.out)*

turns on all warnings



Execution

```
mpixec -n <number of processes> <executable>
```

```
mpixec -n 1 ./mpi_hello
```

 *run with 1 process*

```
mpixec -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 */);
```

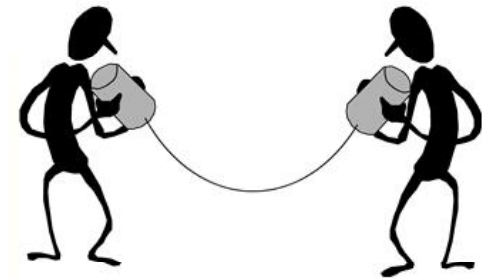
my rank
(the process making this call)

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.

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 */);
```

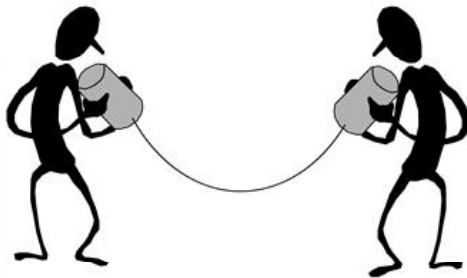


Data types

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 */);
```



Message matching

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

MPI_Send

src = q



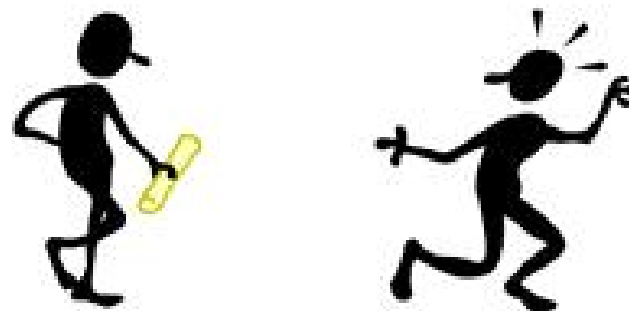
MPI_Recv

dest = r

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

Receiving messages

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



status_p argument

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

MPI_Status*



MPI_Status* status;

status.MPI_SOURCE

status.MPI_TAG

MPI_SOURCE

MPI_TAG

MPI_ERROR

How much data am I receiving?

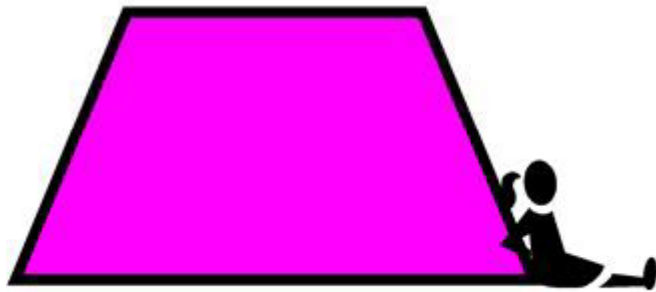
```
int MPI_Get_count(  
    MPI_Status* status_p /* in */,  
    MPI_Datatype type /* in */,  
    int* count_p /* out */);
```



Issues with send and receive

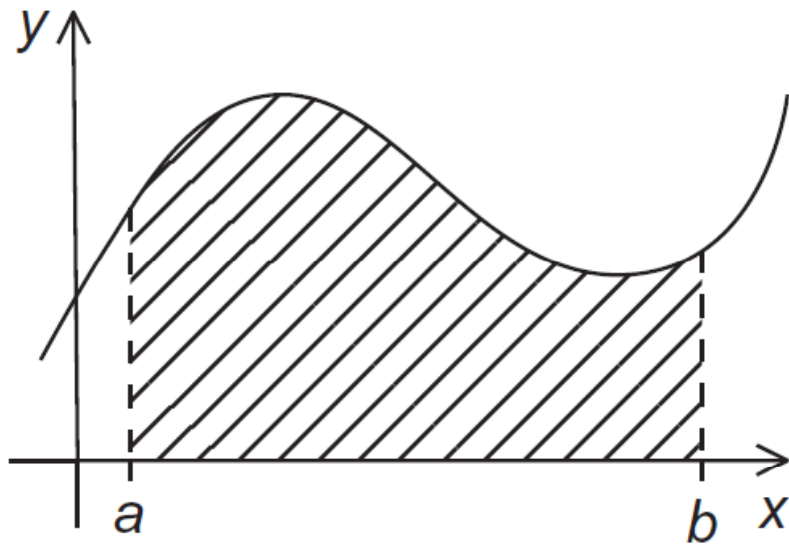
- Exact behavior is determined by the MPI implementation.
- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Know your implementation; don't make assumptions!



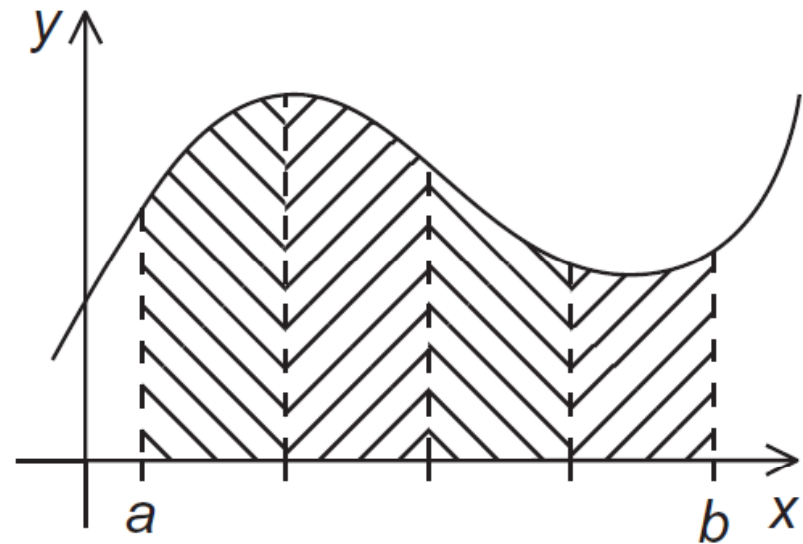


TRAPEZOIDAL RULE IN MPI

The Trapezoidal Rule



(a)



(b)

The Trapezoidal Rule

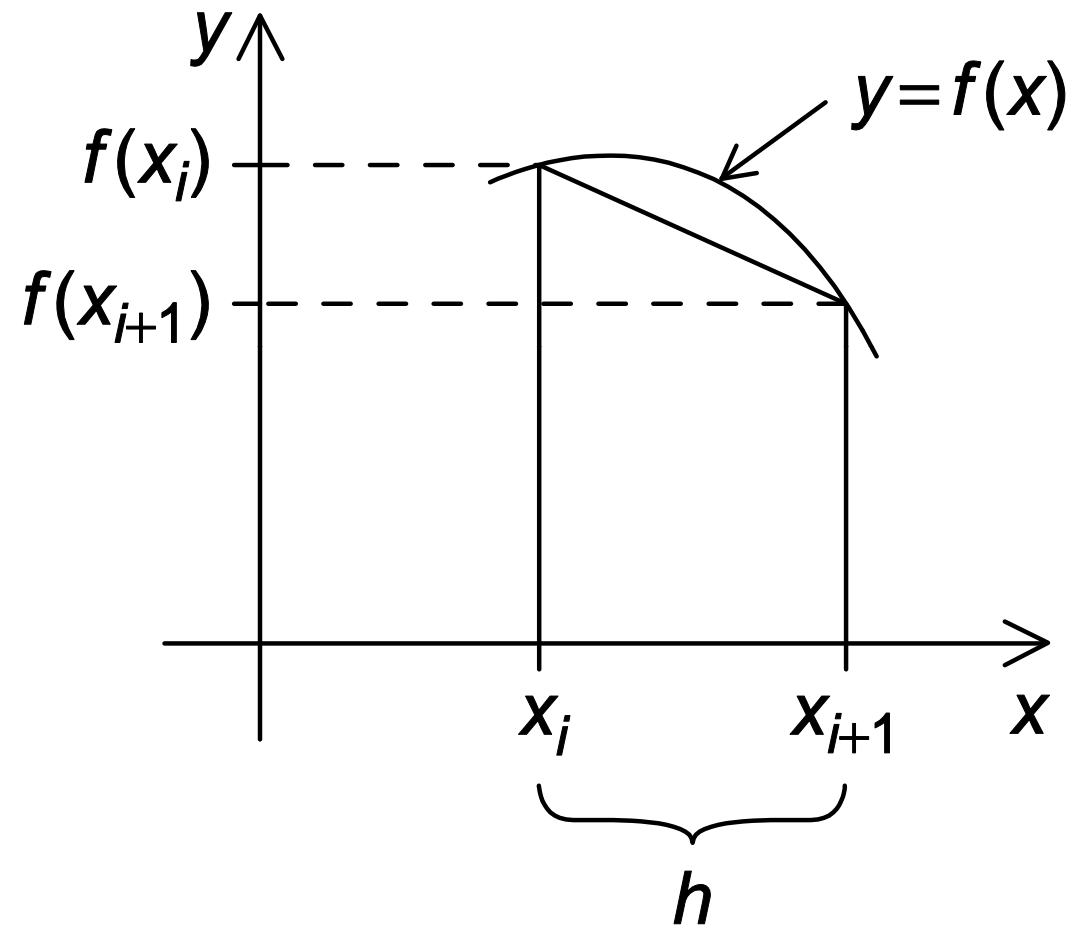
$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

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

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;
```

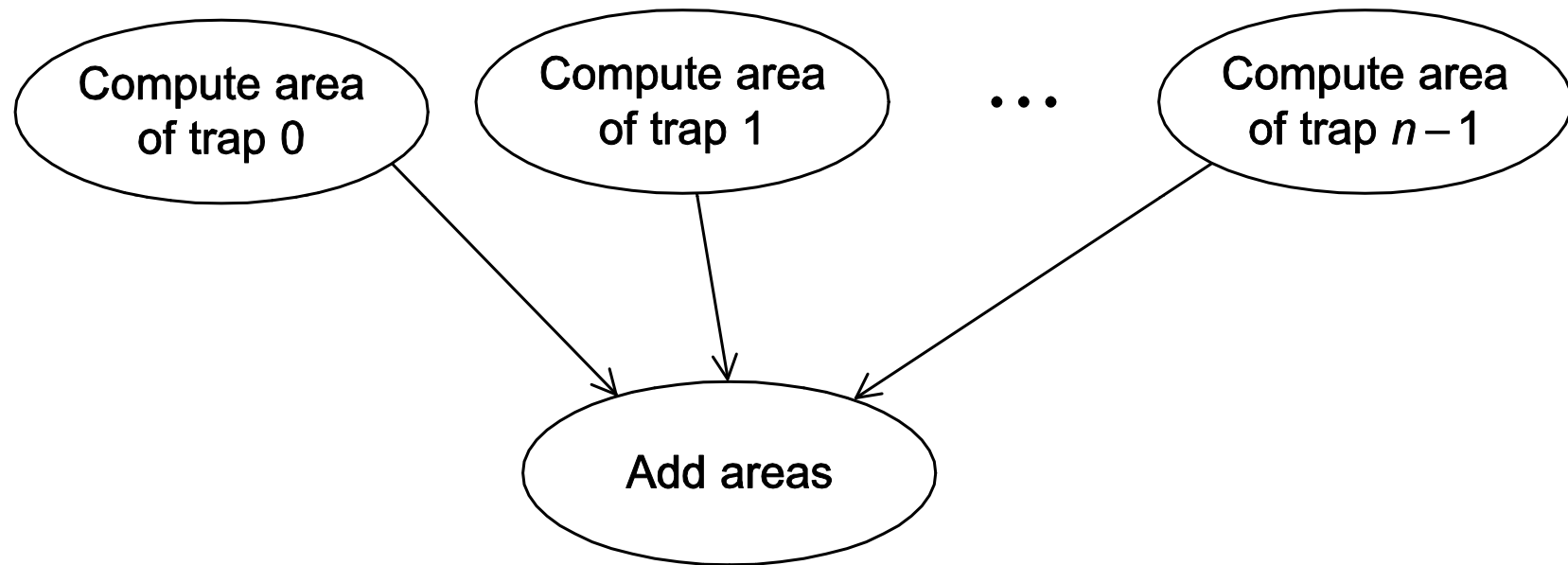
Parallelizing the Trapezoidal Rule

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)

```
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;
```

*Each process just
prints a message.*

```
    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 */
```

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

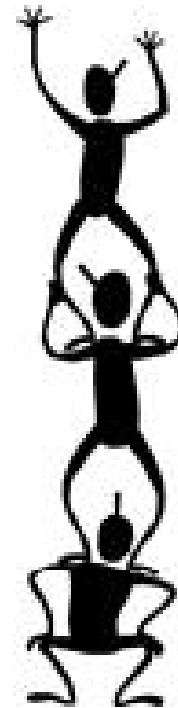
- Most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to `stdin`.
- Process 0 must read the data (`scanf`) and send to the other processes.

```
. . .  
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 */
```

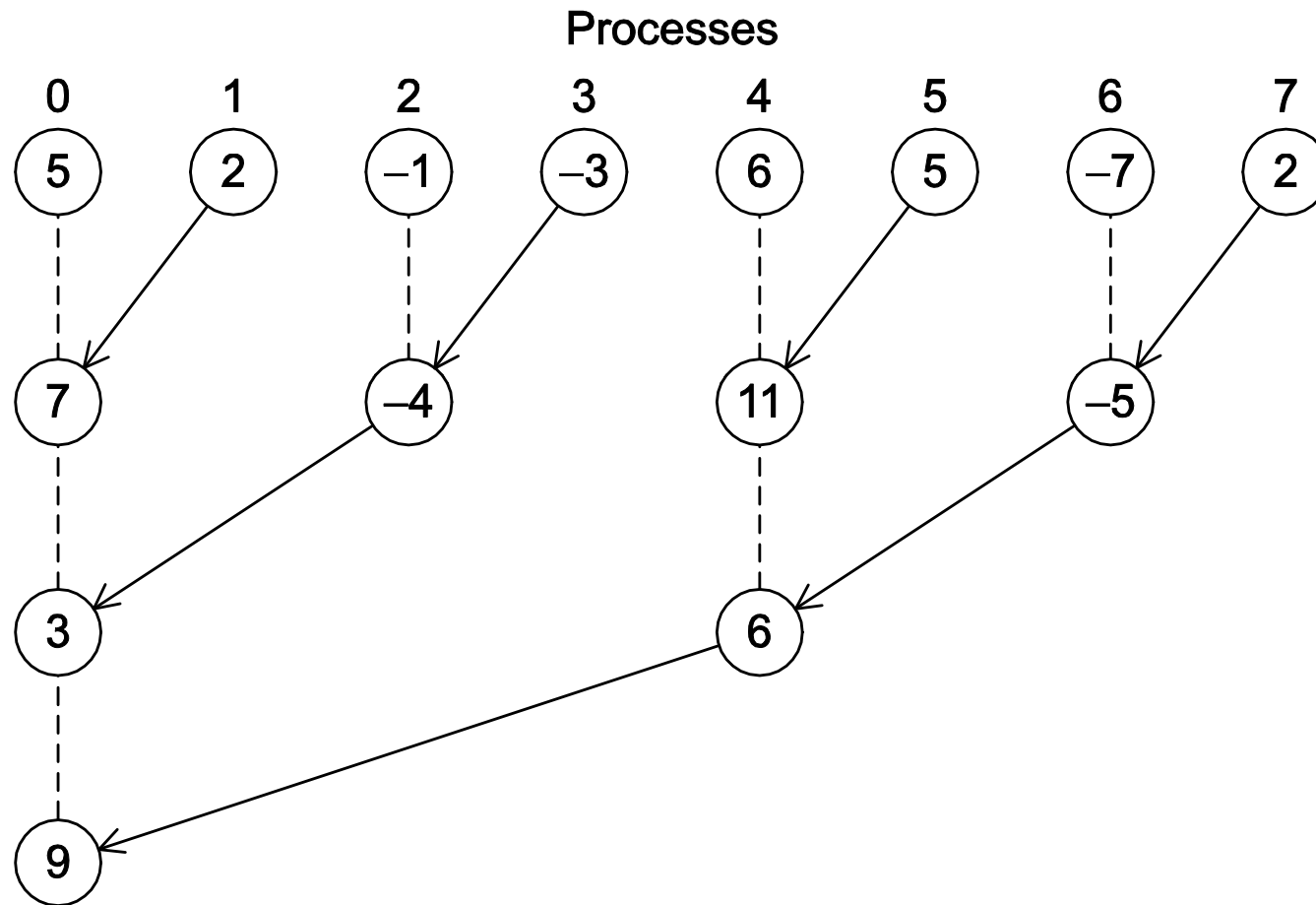
COLLECTIVE COMMUNICATION



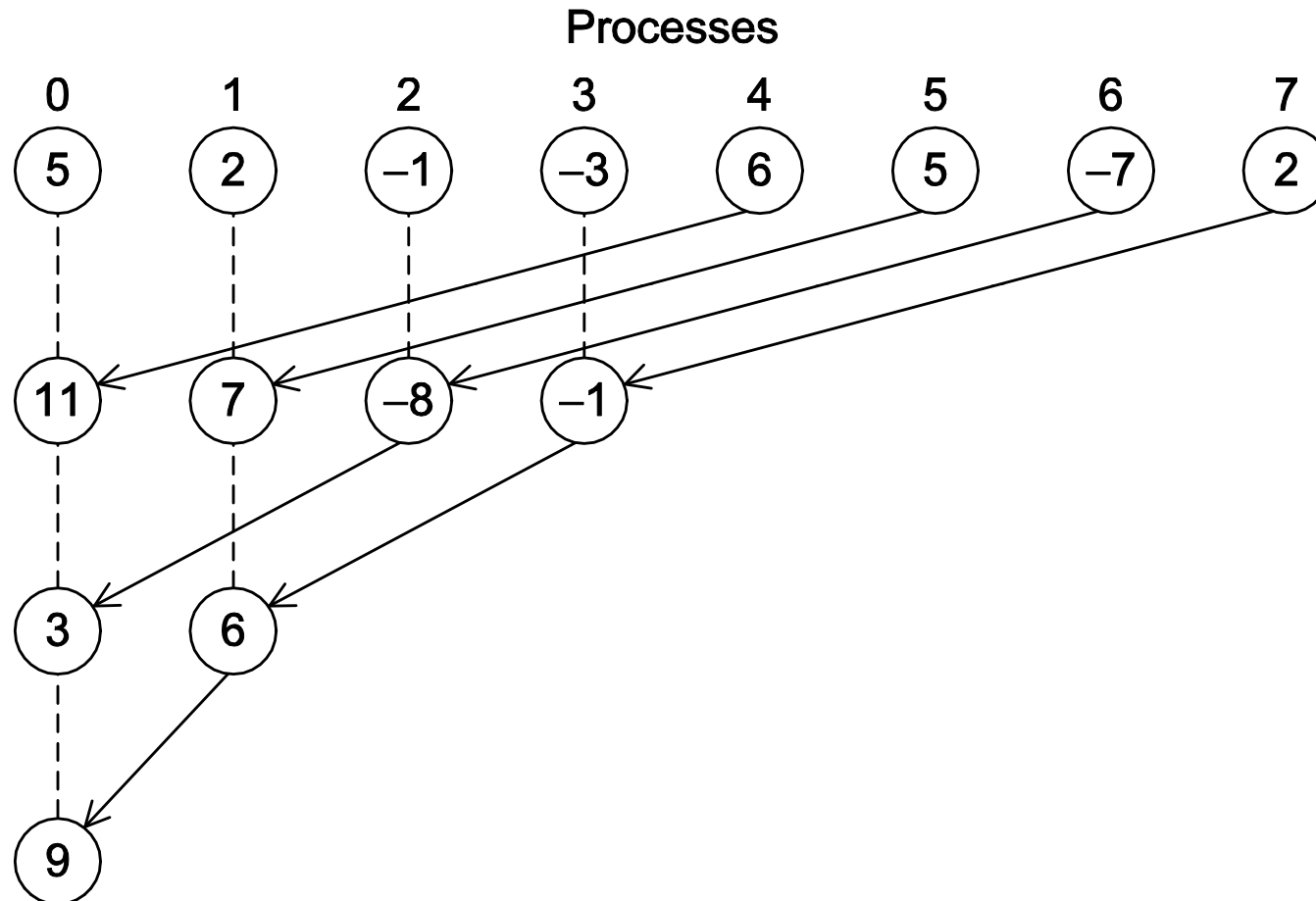
Tree-structured communication

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
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical and
<code>MPI_BAND</code>	Bitwise and
<code>MPI_LOR</code>	Logical or
<code>MPI_BOR</code>	Bitwise or
<code>MPI_LXOR</code>	Logical exclusive or
<code>MPI_BXOR</code>	Bitwise exclusive or
<code>MPI_MAXLOC</code>	Maximum and location of maximum
<code>MPI_MINLOC</code>	Minimum and location of minimum

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.

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, ...)

Multiple calls to MPI_Reduce

Example (2)

- Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0.
- At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be 3, and the value of `d` will be 6.

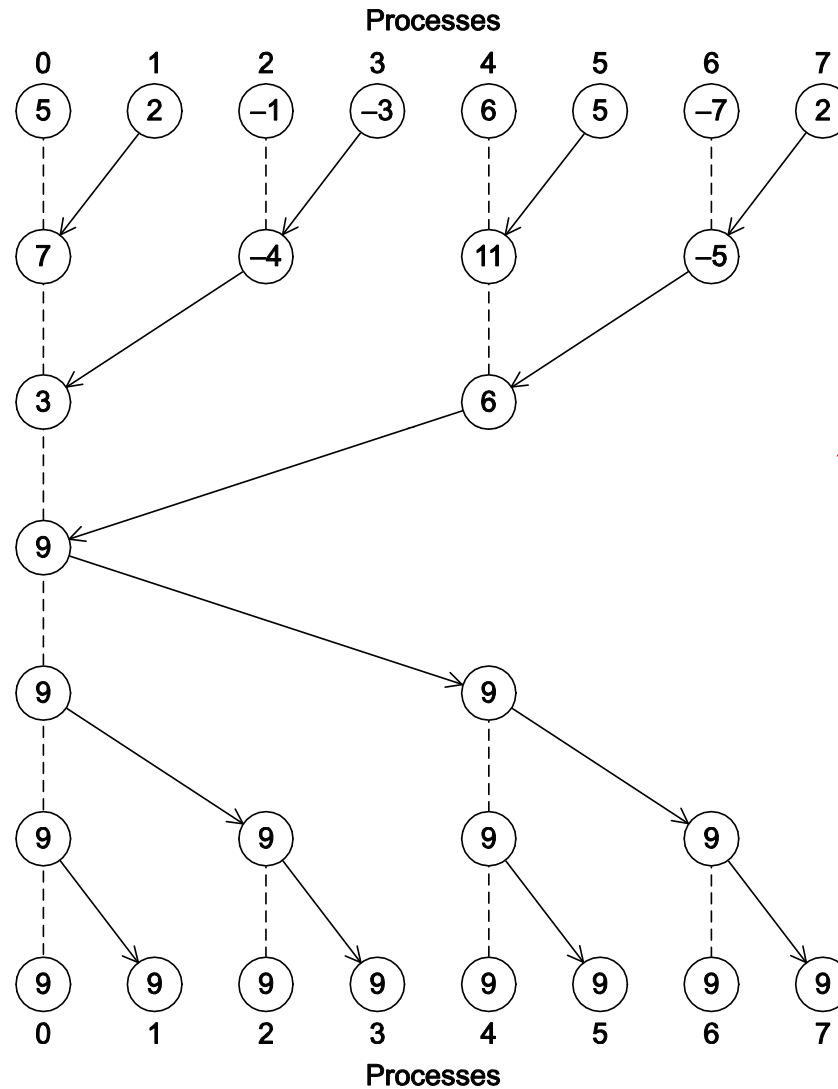
Example (3)

- However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`.
- The order of the calls will determine the matching so the value stored in b will be $1+2+1 = 4$, and the value stored in d will be $2+1+2 = 5$.

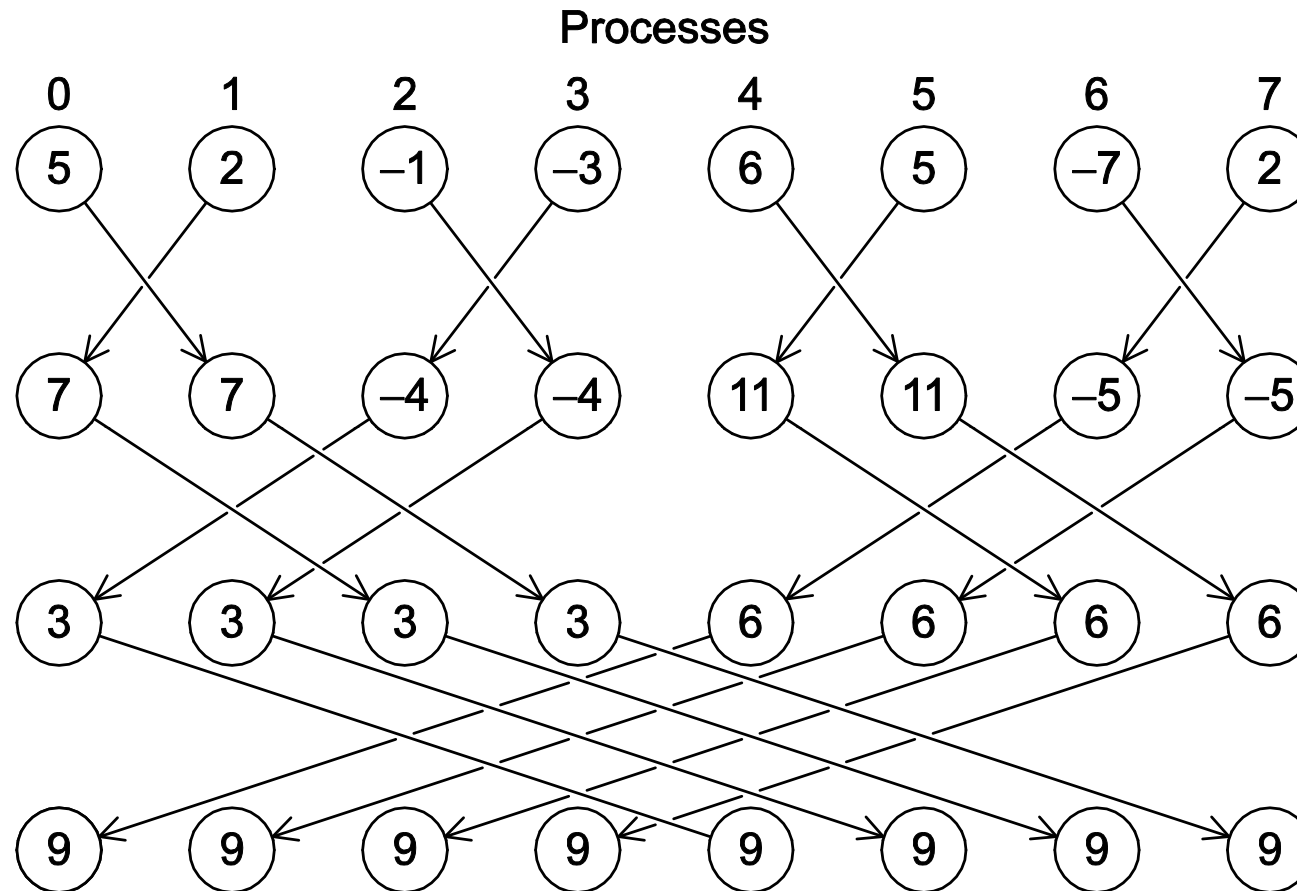
MPI_Allreduce

- 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.*

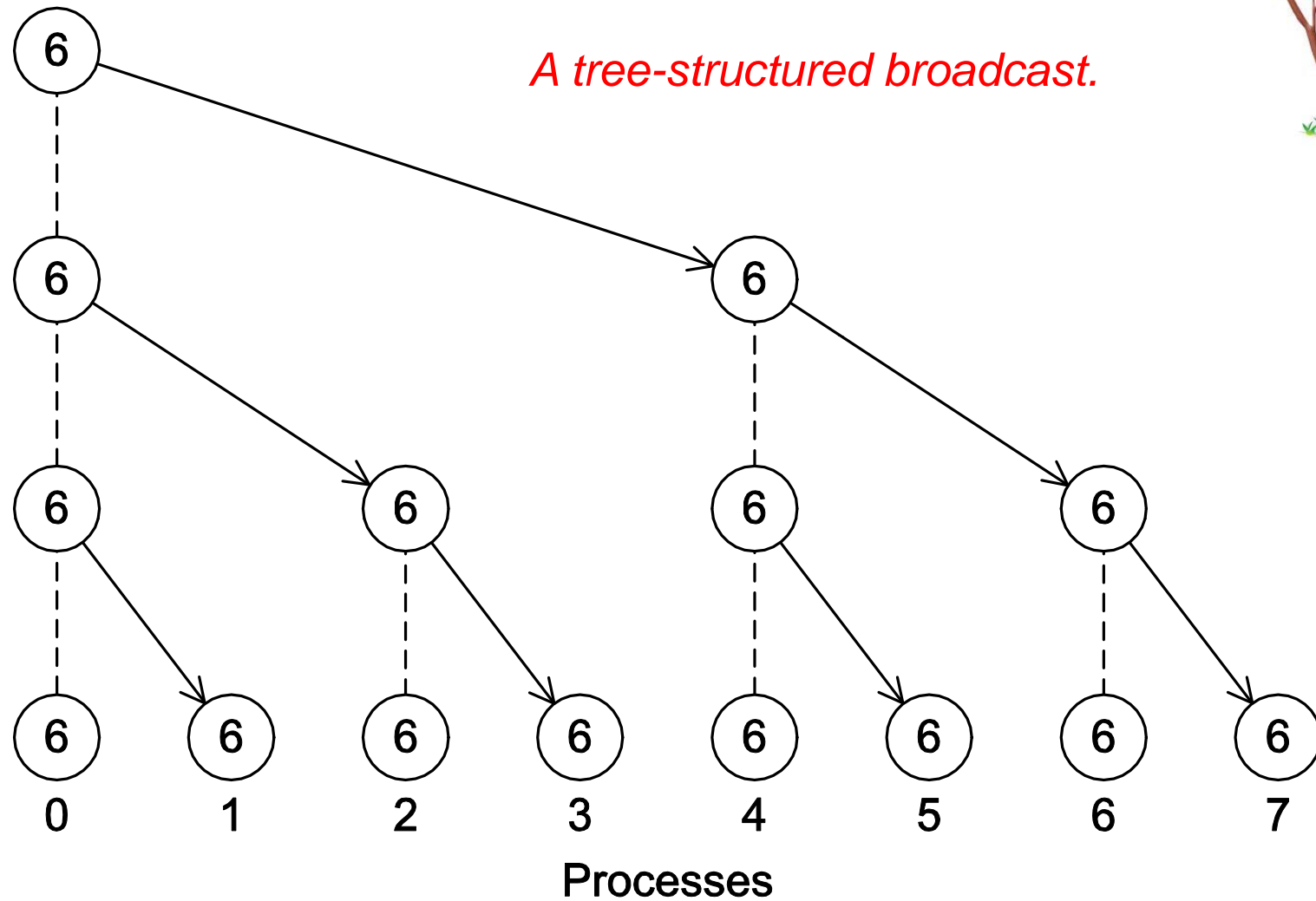


A butterfly-structured global sum.

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 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

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

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 */
```

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 */);
```

Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

\mathbf{x} is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1}$$

i -th component of y

*Dot product of the i th
row of A with x .*

Matrix-vector multiplication

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
x_1
\vdots
x_{n-1}

 $=$

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

Multiply a matrix by a vector

```
/* For each row of A */  
for (i = 0; i < m; i++) {  
    /* Form dot product of ith row with x */  
    y[i] = 0.0;  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]*x[j];  
}
```

Serial pseudo-code

C style arrays

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Serial matrix-vector multiplication

```
void Mat_vect_mult(  
    double A[] /* in */,  
    double x[] /* in */,  
    double y[] /* out */,  
    int m /* in */,  
    int n /* in */) {  
    int i, j;  
  
    for (i = 0; i < m; i++) {  
        y[i] = 0.0;  
        for (j = 0; j < n; j++)  
            y[i] += A[i*n+j]*x[j];  
    }  
} /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(  
    double    local_A[]    /* in  */,  
    double    local_x[]    /* in  */,  
    double    local_y[]    /* out */,  
    int        local_m      /* in  */,  
    int        n            /* in  */,  
    int        local_n      /* in  */,  
    MPI_Comm   comm         /* in  */) {  
    double* x;  
    int local_i, j;  
    int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

```
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
              x, local_n, MPI_DOUBLE, comm);

for (local_i = 0; local_i < local_m; local_i++) {
    local_y[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
}
free(x);
} /* Mat_vect_mult */
```



MPI DERIVED DATATYPES

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

$\{(\text{MPI_DOUBLE}, 0), (\text{MPI_DOUBLE}, 16), (\text{MPI_INT}, 24)\}$

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 */
```

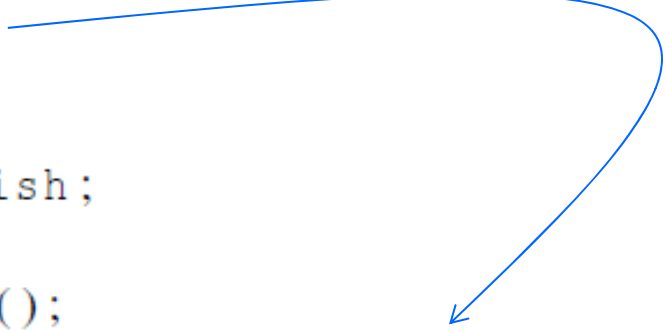


PERFORMANCE EVALUATION

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);
```



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.

```
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 processes increase.



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 $p = \text{comm sz processes}$.
- 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:

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

- Odd phases, compare swaps:

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

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

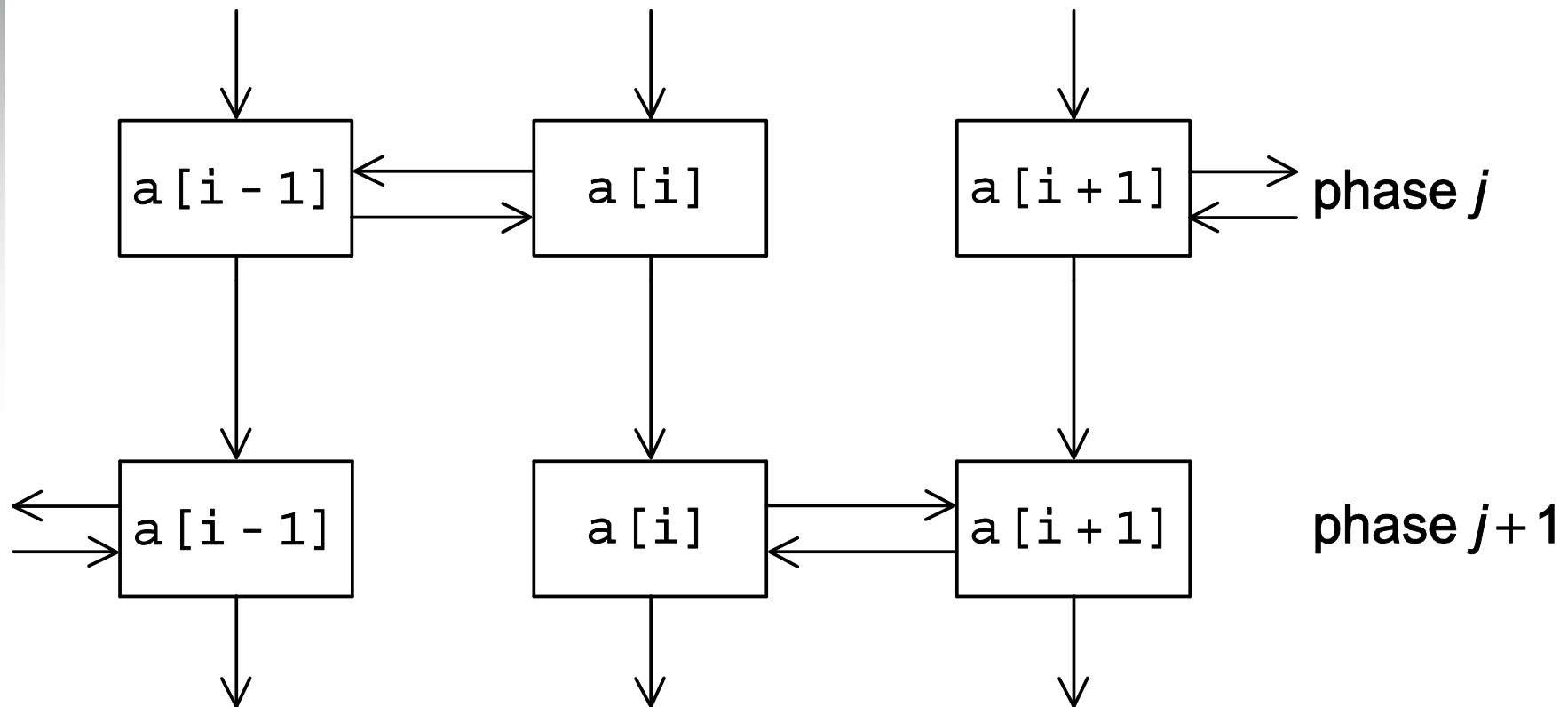
Even phase: compare-swap (5,3) and (9,4)
getting the list 3, 5, 4, 9

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.

(see pseudo-code)

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.

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);  
}
```

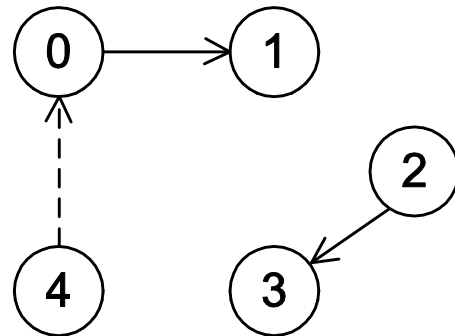
MPI_Sendrecv

- 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.

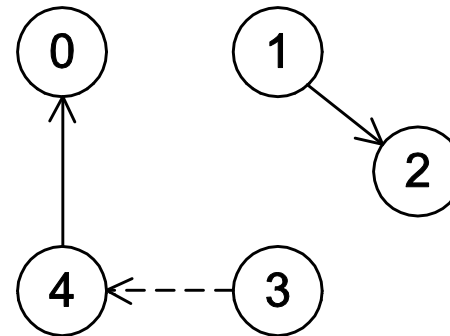
MPI_Sendrecv

```
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 */ );
```

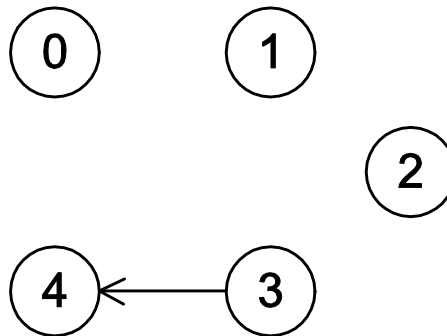
Safe communication with five processes



Time 0



Time 1



Time 2

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)

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.