High-Performance Computing

– Distributed Memory Programming with MPI –

Christian Plessl

High-Performance IT Systems Group

Paderborn University

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

version 1.2.3 2017-11-20

Paderborn Center for Parallel Computing

Outline

- **Overview of MPI**
- **Basic MPI**
    - Hello world
    - basic MPI functions
    - example: trapezoidal rule
- **Intermediate MPI**
    - collective communication
    - derived datatypes
    - performance measurements
    - example: parallel sorting
    - safety in MPI programs
- **Advanced MPI (in future supplement to these slides)**
    - non-blocking peer-to-peer communication
    - one-sided communication
    - I/O

The Message Passing Interface (MPI)

- **Messages are transported between processes by explicit cooperation of sending and receiving processes**
- **Increasing popularity in the 1980s and early 90s**
  - numerous vendor-specific libraries, but no standard
  - limited portability of code between different machines
  - differences in features, efficiency, price

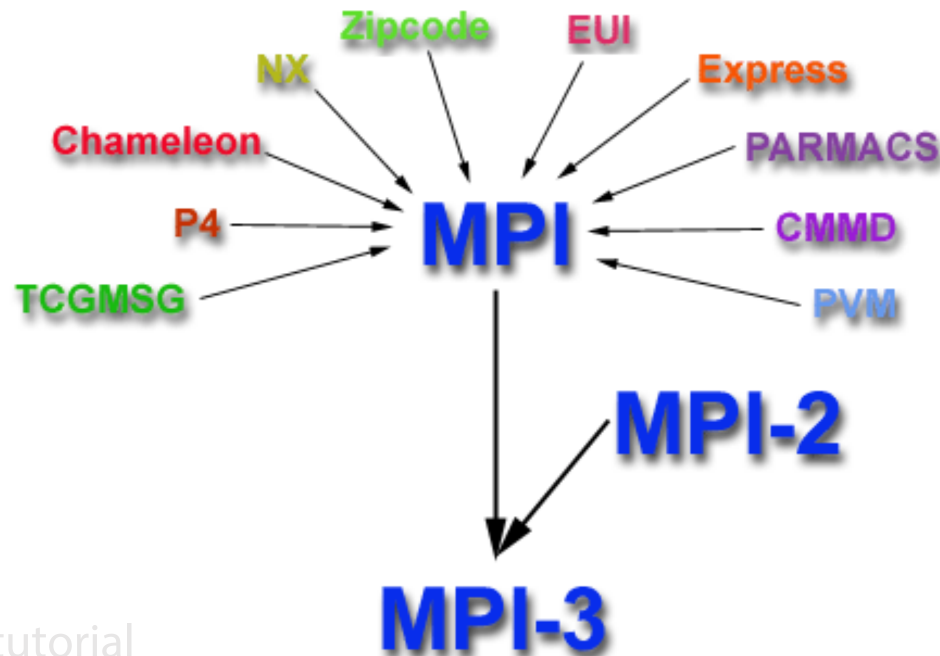


image: LLNL MPI tutorial

The Message Passing Interface (MPI)

- **Need for a common standard recognized in 1992**
- **First drafts in 1992, final standard of MPI-1.0 in 1994**
- **Today, MPI the most widely used standard for message passing**
  - MPI is a specification for users and developers of a message passing library (not a library itself)
  - defines C and Fortran interfaces
- **There exist several widely used implementations, e.g.**
  - OpenMPI, Intel MPI, MPICH, MVAPICH, . . .
  - vendors can innovate in the implementation, e.g. offloading part of MPI functions to network cards (e.g. broadcast, barrier)
- **Standard has been extended over time**
  - initially rather simple
  - over time, more complex features added, latest MPI-3.1 standard has 800+ pages

Hello World!

```
/*  This is a short program */
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("hello, world\n");

   return 0;
}
```

Identifying MPI processes

- **Common practice to identify processes by nonnegative integer numbers, denoted as ranks**
  - p processes are numbered 0, 1, 2, .. p-1
  - rank 0 is typically reserved for the "master process"
- **Processes can query their own rank and the total number of ranks at runtime**

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

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

```
mpiexec  -n  1  ./mpi_hello
```

```
mpiexec  -n  4  ./mpi_hello
```

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 (in this lecture, Fortran interface available too)**
  - need main function
  - include headers for standard library functions, e.g., stdio.h, string.h, etc.
- **MPI-specific functions declared in 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
- **Compilation and linking with MPI-specific libraries**
  - simplified with mpicc compiler wrapper
  - wrapper adds required libraries automatically

MPI Program Structure

- **MPI_Init**
  - needs to be called by all ranks before using any MPI functions
  - tells MPI to do all the necessary setup

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

- **MPI_Finalize**
  - needs to be called by all ranks at the end of the program
  - frees up any resources allocated for this program

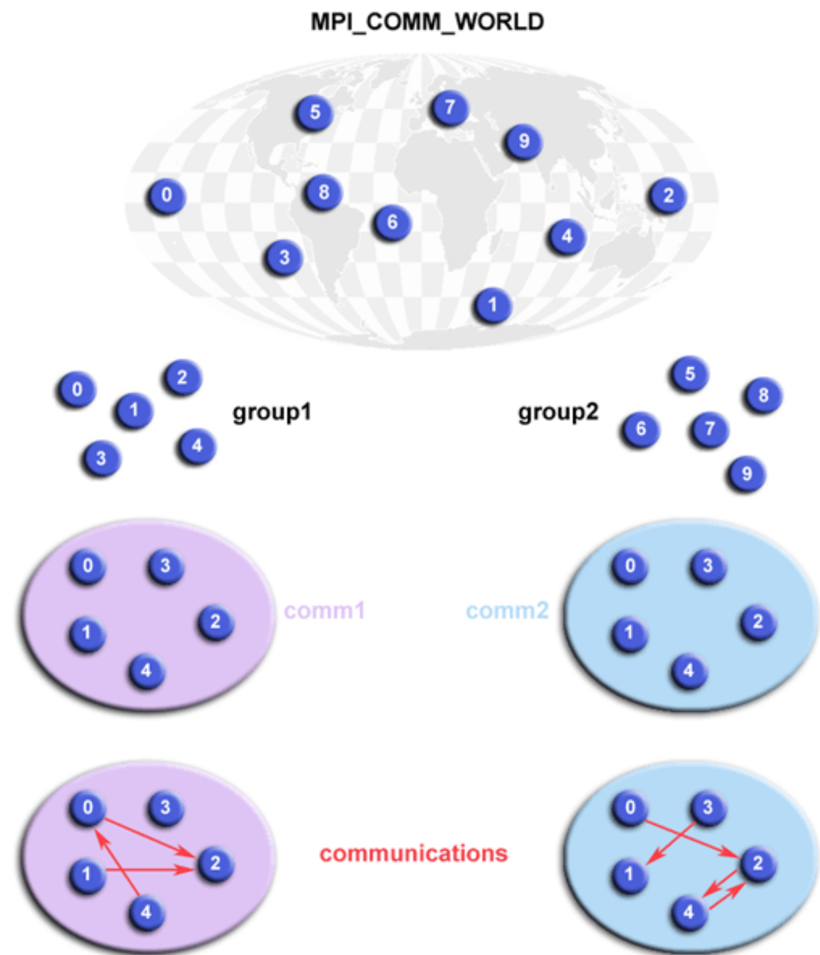
```
int MPI_Finalize(void);
```

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 communicator is a group of processes that may send messages to each other**
- **MPI_Init creates a default communicator**
    - consists of all the processes created when the program is started
    - called MPI_COMM_WORLD
- **Additional communicators and groups can be created at runtime**
    - simplifies repeated communication within subset of nodes (e.g. all even or all odd nodes)
    - for our basic needs typically not required



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 */);
```

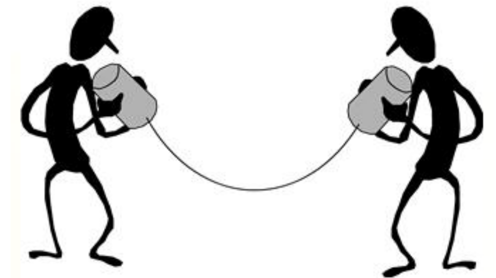
my rank
(the process making this call)

SPMD (Single-Program Multiple Data)

- **The code for all processes (ranks) is specified with *one* program**
- **Advantages**
  - simplified compilation, deployment and execution of code on many processors
  - tailoring the execution to the available number of processes (ranks) is happens at runtime, not compile time
- **Rank-specific code implemented by conditional code (if-else constructs)**
- **Rank 0 is different**
  - designated Master process
  - has access to standard input
  - (in the hello world example: receives messages and prints them while the other processes do the work)

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 */);
```

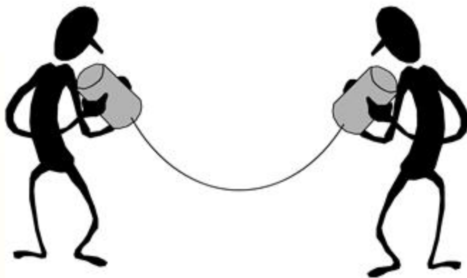


Data Types

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);
```

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

typically MPI_COM_WORLD

MPI point-to-point communication requires that *all* of these "Message Envelope" parameters match!

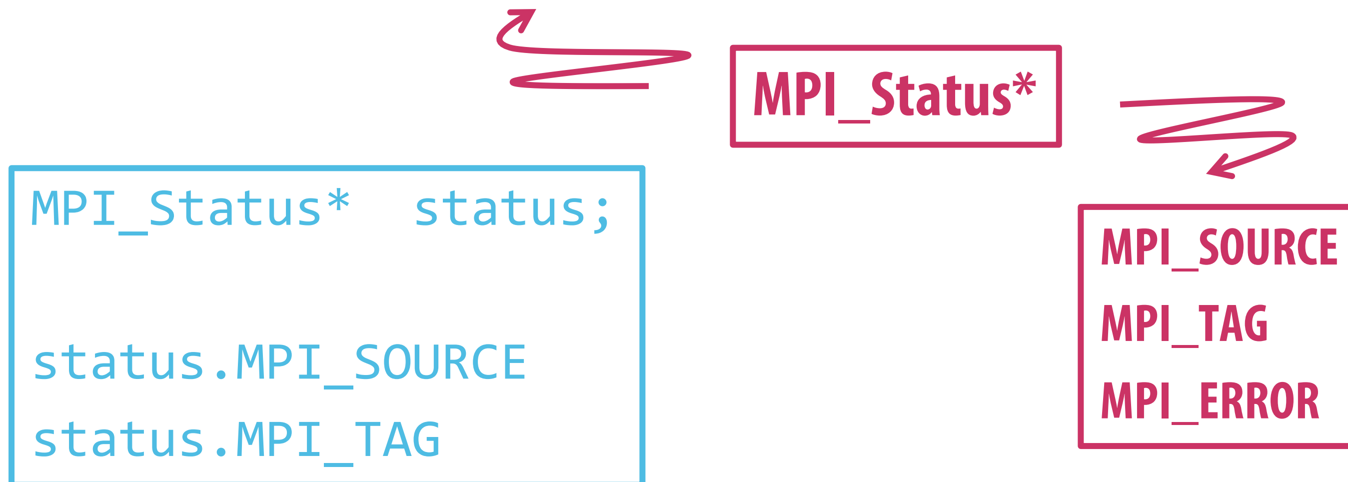
Receiving Messages

- **A receiver can also receive a message without knowing ..**
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message
- **.. by specifying wildcards for source and tag**
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

Determine Source, Tag and Length

When using wildcards (MPI_ANY_SOURCE or MPI_ANY_TAG) the sender can use the MPI_Status* return value to identify the source, tag of a message

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

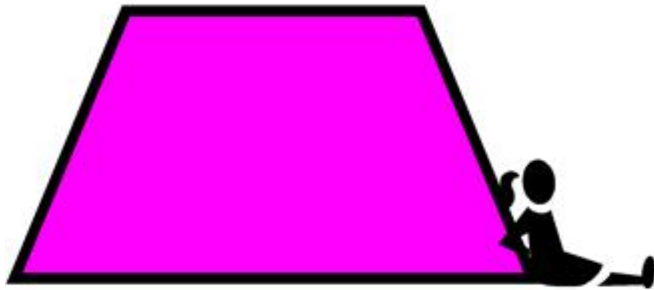


How Much Data am I Receiving?

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

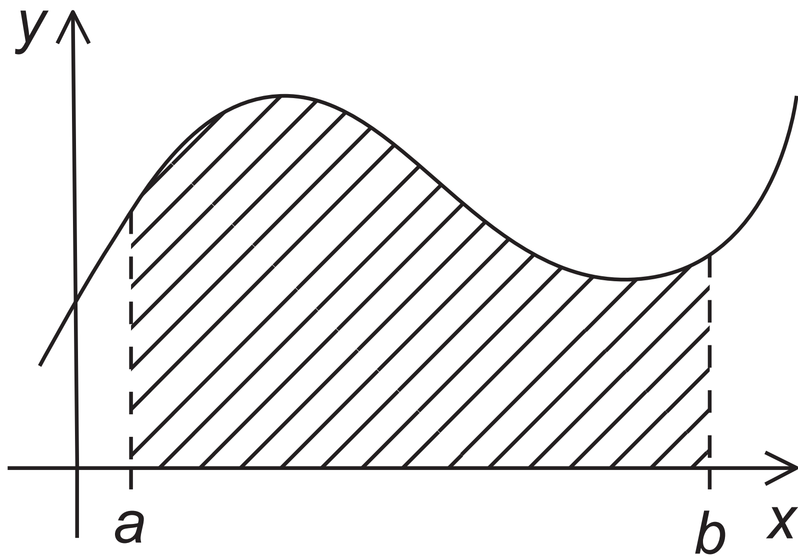
Issues with Send and Receive

- 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



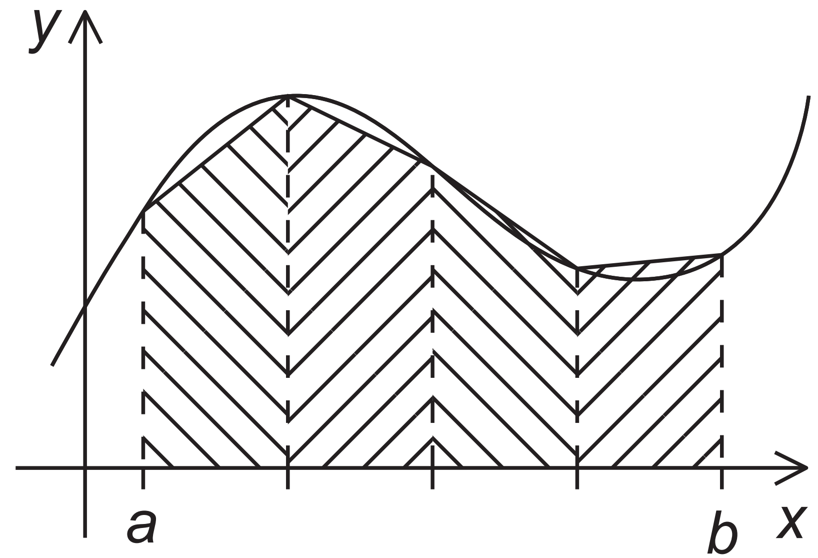


Figure 3.3

One Trapezoid

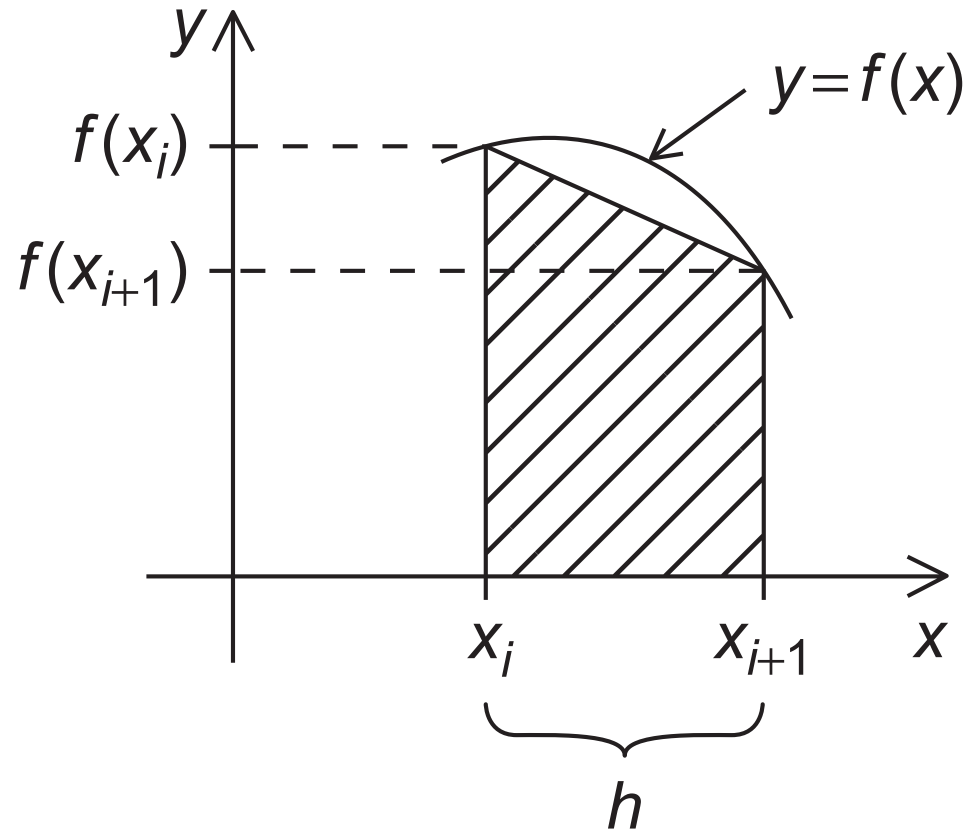


Figure 3.4

a (a) b x a (b) b x

The Trapezoidal Rule

(a) (b)

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

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

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

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

Pseudo-Code for a Serial Program

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

Parallelizing the Trapezoidal Rule

- **Apply Foster's approach**
  - Partition problem solution into tasks
  - Identify communication channels between tasks
  - Aggregate tasks into composite tasks
  - 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

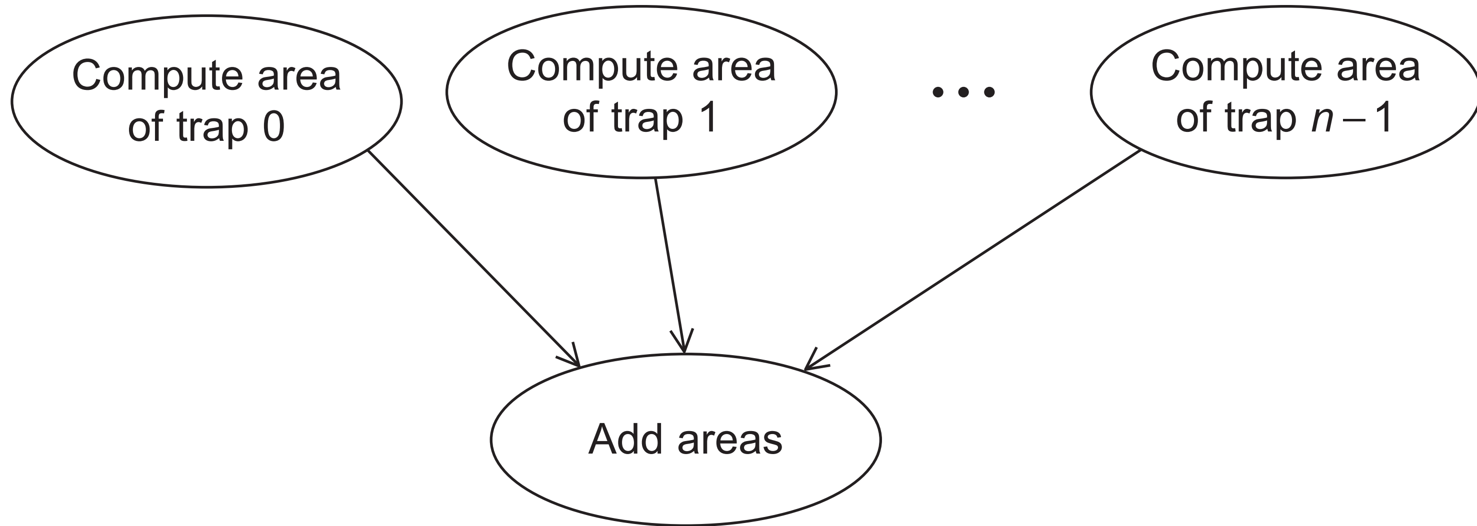


Figure 3.5

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;

   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 */
```

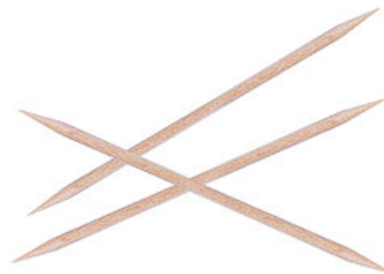
Each process just prints a message

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?
```

Example output (could be any other permutation)

Non-deterministic 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_input(my_rank, comm_sz, &a, &b, &n);

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

all ranks run this function

Example: passing the input data to all processes

Function for Reading User Input

```
1   void Get_input(
2         int      my_rank   /* in  */,
3         int      comm_sz   /* in  */,
4         double*  a_p       /* out */,
5         double*  b_p       /* out */,
6         int*     n_p       /* out */) {
7      int dest;
8
9      if (my_rank == 0) {
10        printf("Enter a, b, and n\n");
11        scanf("%lf %lf %d", a_p, b_p, n_p);
12        for (dest = 1; dest < comm_sz; dest++) {
13           MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14           MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15           MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16        }
17     } else { /* my_rank != 0 */
18        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19              MPI_STATUS_IGNORE);
20        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21              MPI_STATUS_IGNORE);
22        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23              MPI_STATUS_IGNORE);
24     }
25  }  /* Get_input */
```

This Could be the End

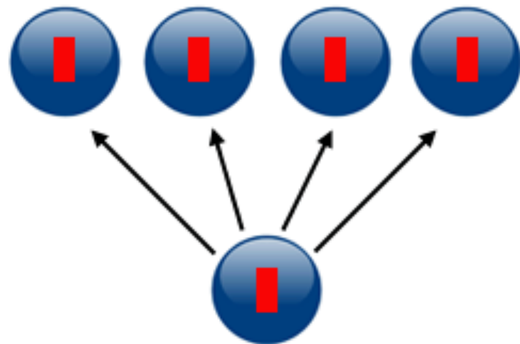
- **Essentially, we could stop discussing MPI here**
- **Everything can be built on top of point-to-point send and receive functions**
- **What is missing**
  - higher-level communication abstractions for convenience
  - efficient one-to-many, many-to-one, and many-to-many communication
  - functions for safe (deadlock free) operation
  - one-sided communication
  - efficient file I/O

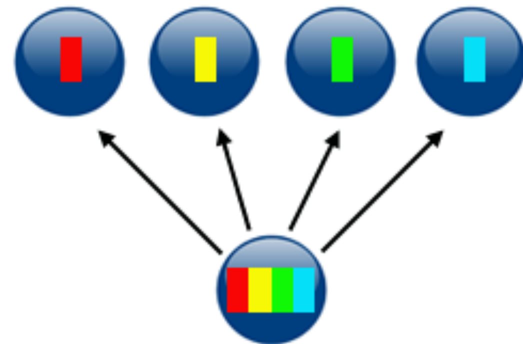
MPI Collective Communication Routines

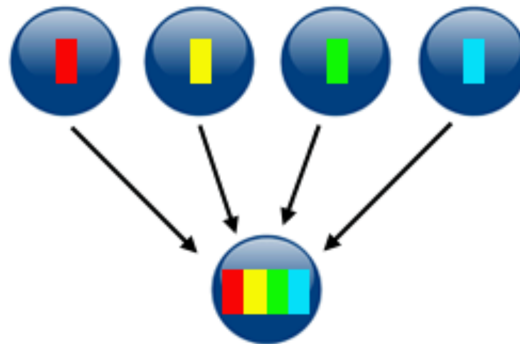
- **MPI Collectives perform operations by involving all tasks available within one communicator**

- **Type of collective operations**
  - synchronization: let all processes wait until all members of the group have reached a synchronization point (barrier)
  - data movement: one-to-many and many-to-many communications (broadcast, scatter, gather, all to all)
  - collective computation (reductions): one member of the group collects data from all other members and performs an operation on the data (min, max, add, multiply, . . .)

Collections Visual Overview







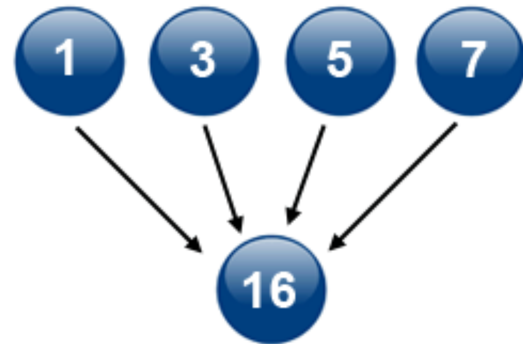


image: LLNL MPI tutorial

Reduction: Tree-Structured Global Sum

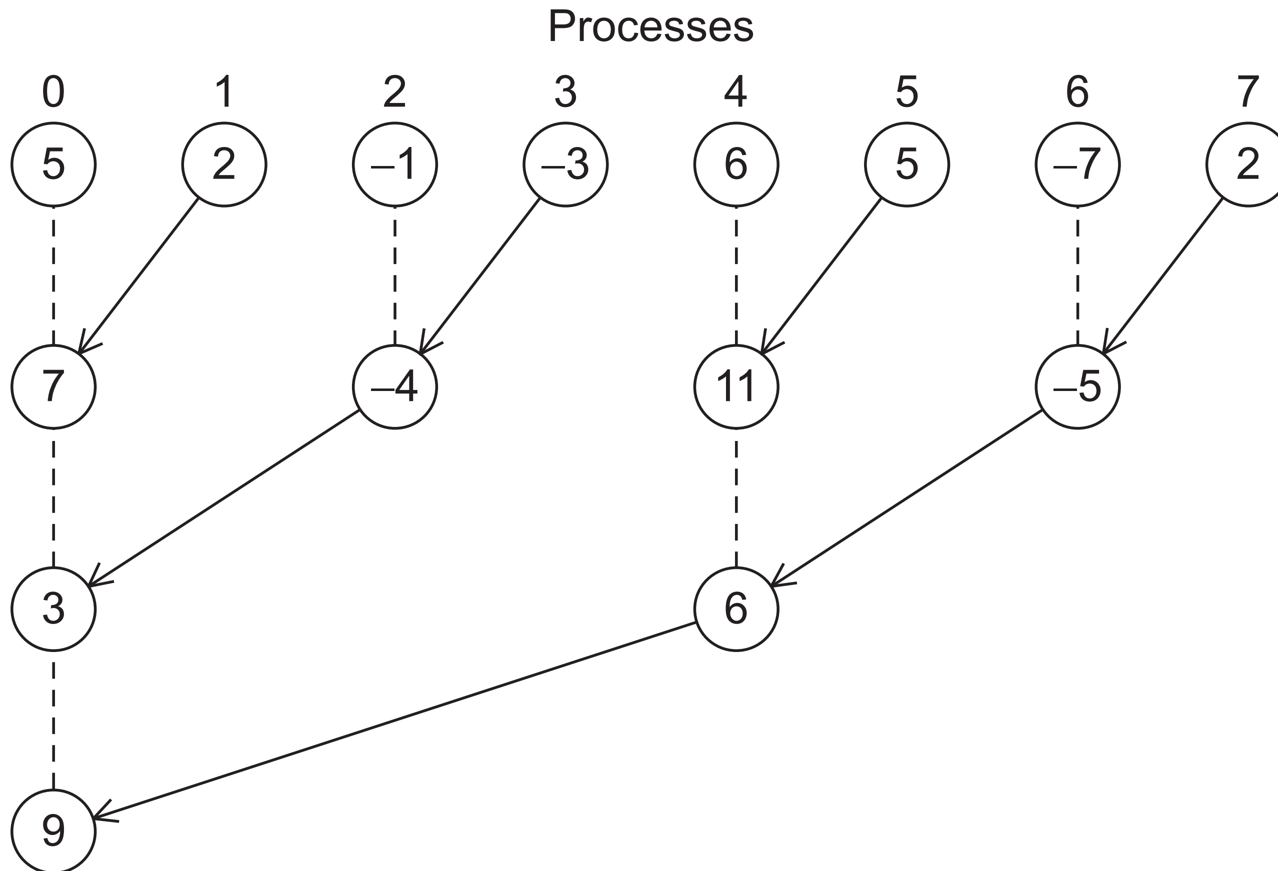


Figure 3.6

An Alternative Tree-Structured Global Sum

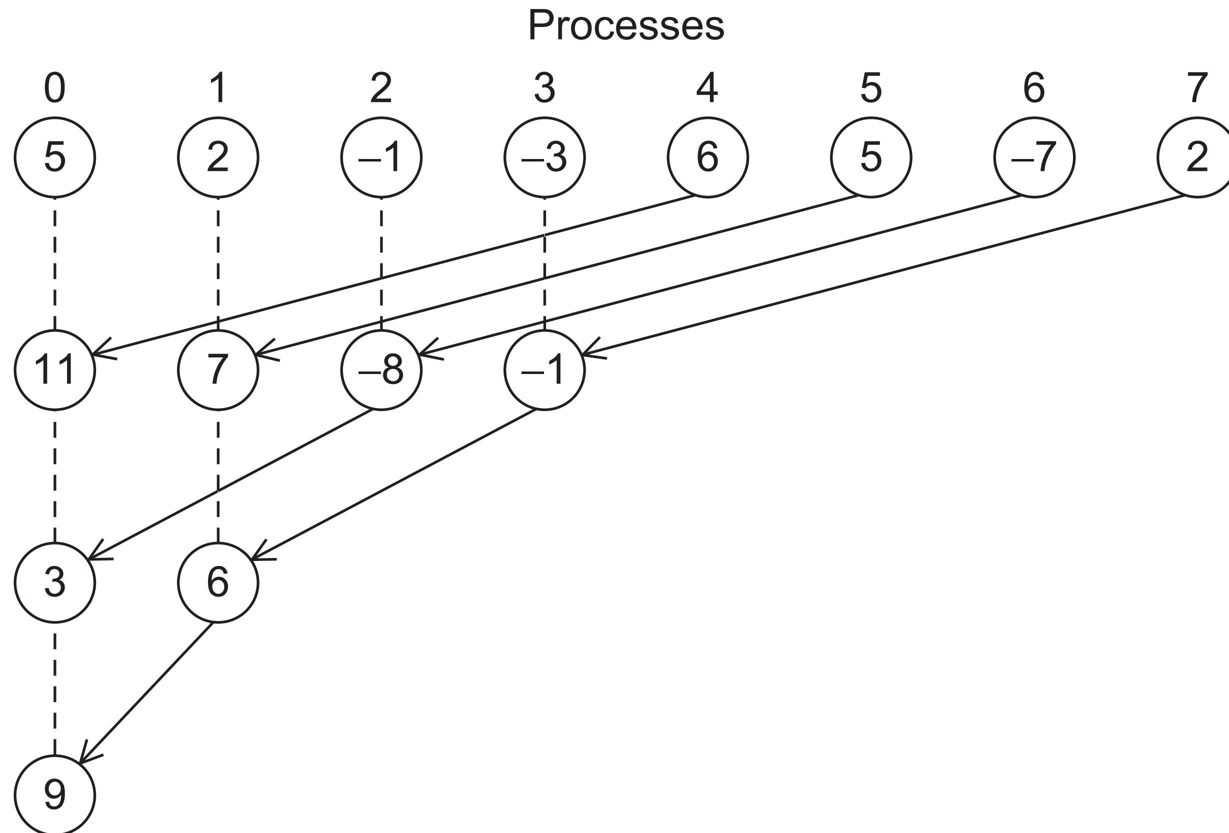


Figure 3.7

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 function signature

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

example: sum scalar value (local_int) from each rank

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

example: element-wise summation of arrays from each rank

MPI_Reduce Illustration

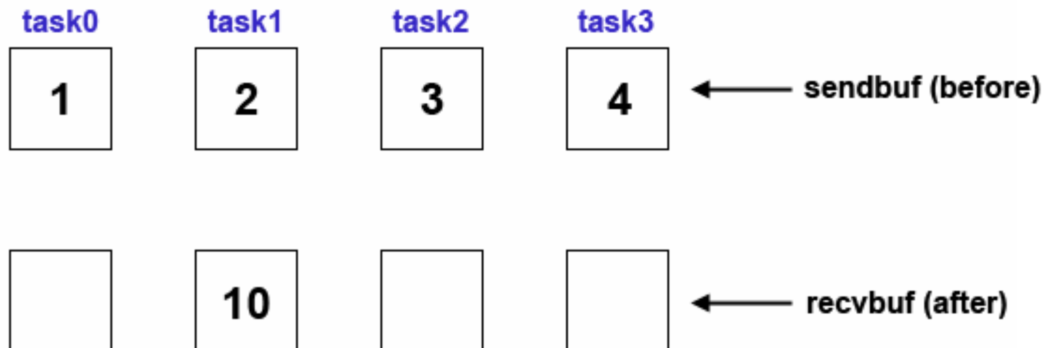


image: LLNL MPI tutorial

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 vs. Point-to-Point Communications

- **All the processes in the communicator must call the same collective function**
  - e.g. 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 will cause the program to hang or crash
  - arguments passed by each process to an MPI collective must be "compatible", e.g. 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 will cause the program to hang or crash
  - 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
- **Send and receive matching**
  - point-to-point communication: matched on the basis of tags and communicators
  - collective communications: don't use tags, matching on the basis of the communicator and the order in which they're called

Example

- **Collectives are matched by order in which they are called (not tags)**
  - example: MPI_ADD reduction, with rank 0 as destination
  - after executing these statements value of b=1+2+1 and d=2+1+2

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, ...)`

memory location of destination used by senders is irrelevant

- **Using the same buffer for both input and output is illegal and may result in invalid results (aliasing)**

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

- **MPI has special functions for sending and receiving using a single buffer (e.g. MPI_Sendrecv_replace)**

MPI_Allreduce

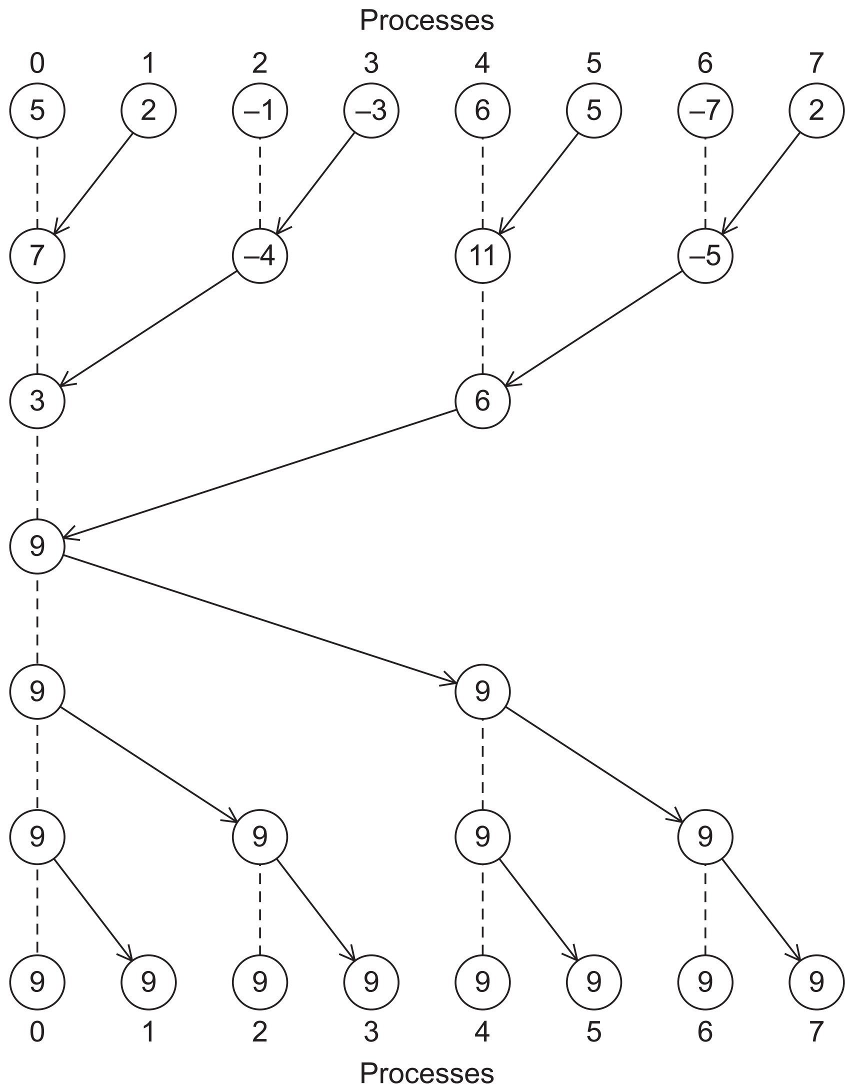
- **Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation**

```
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  */);
```

MPI_Allreduce function signature

- **Essentially the same function as MPI_Reduce but without dest_process argument, because all process shall receive the result**

Naïve implementation of MPI_Allreduce



A global sum followed by distribution of the result.

Figure 3.8

Efficient Implementation of MPI_Allreduce

Processes

0 1 2 3 4 5 6 7

5 2 –1 –3 6 5 –7 2

7 7 –4 –4 11 11 –5 –5

3 3 3 3 6 6 6 6

9 9 9 9 9 9 9 9

A butterfly-structured global sum

Figure 3.9

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     */);
```

MPI_Bcast function signature

- **For the sending process data_p is an input, for all other processes an output**

A tree-structured broadcast

6

6 6

6 6 6 6

6 6 6 6 6 6 6 6

0 1 2 3 4 5 6 7

Processes

Figure 3.10

MPI_Bcast Illustration

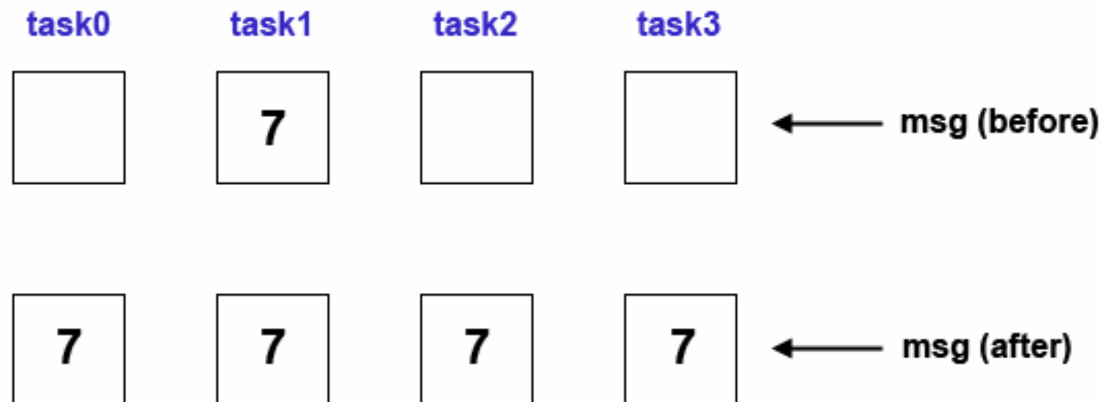


image: LLNL MPI tutorial

Recap: Function for Reading User Input

```
1   void Get_input(
2         int      my_rank   /* in  */,
3         int      comm_sz   /* in  */,
4         double*  a_p       /* out */,
5         double*  b_p       /* out */,
6         int*     n_p       /* out */) {
7      int dest;
8
9      if (my_rank == 0) {
10        printf("Enter a, b, and n\n");
11        scanf("%lf %lf %d", a_p, b_p, n_p);
12        for (dest = 1; dest < comm_sz; dest++) {
13           MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14           MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15           MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16        }
17     } else { /* my_rank != 0 */
18        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19              MPI_STATUS_IGNORE);
20        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21              MPI_STATUS_IGNORE);
22        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23              MPI_STATUS_IGNORE);
24     }
25  }  /* Get_input */
```

point-to-point communication

Implementation of Get_input using MPI_Bcast

```
1   void Get_input(
2           int      my_rank  /* in  */,
3           int      comm_sz  /* in  */,
4           double*  a_p      /* out */,
5           double*  b_p      /* out */,
6           int*     n_p      /* out */) {
7
8       if (my_rank == 0) {
9           printf("Enter a, b, and n\n");
10          scanf("%lf %lf %d", a_p, b_p, n_p);
11      }
12      MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13      MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14      MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15  }   /* Get_input */
```

broadcast: simpler, more efficient

Data Distributions

- **Goal: We want to efficiently execute operations on matrices or vectors in a distributed memory machine**
- **Example: vector summation**

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

- **Serial implementation**

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

Distribution of Data to Processes

- How to partition and distribute data to processes
- Example: partitioning a 12 component vector to 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

- 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

- **Once the data is distributed, the implementation of the parallel code is straight forward**

```
1   void Parallel_vector_sum(
2         double  local_x[]  /* in  */,
3         double  local_y[]  /* in  */,
4         double  local_z[]  /* out */,
5         int     local_n    /* in  */) {
6      int local_i;
7
8      for (local_i = 0; local_i < local_n; local_i++)
9         local_z[local_i] = local_x[local_i] + local_y[local_i];
10  }  /* Parallel_vector_sum */
```

A parallel implementation of vector addition

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  */);
```

MPI_Scatter function signature

- **Performs only block distribution, works only if number of elements is evenly divisible by number of processes**
- **Note: send_count and recv_count is size of elements being sent to each process, not total size of data**

MPI_Scatter Illustration

MPI_Scatter

Sends data from one task to all other tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered

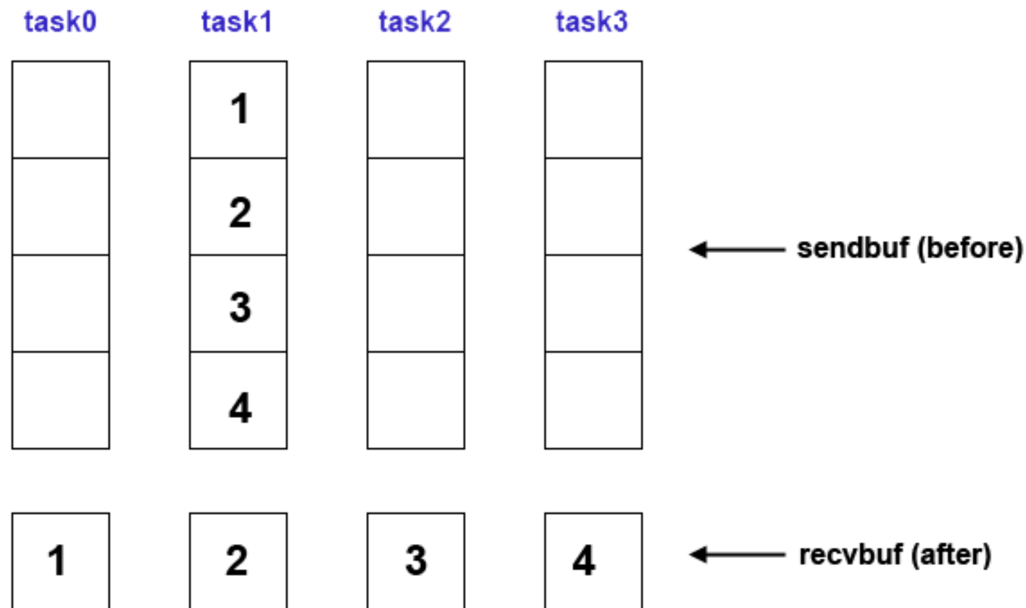


image: LLNL MPI tutorial

Reading and Distributing a Vector

```
 1  void Read_vector(
 2        double    local_a[]   /* out */,
 3        int       local_n     /* in  */,
 4        int       n           /* in  */,
 5        char      vec_name[]  /* in  */,
 6        int       my_rank     /* in  */,
 7        MPI_Comm  comm        /* in  */) {
 8
 9     double* a = NULL;
10     int i;
11
12     if (my_rank == 0) {
13        a = malloc(n*sizeof(double));
14        printf("Enter the vector %s\n", vec_name);
15        for (i = 0; i < n; i++)
16           scanf("%lf", &a[i]);
17        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18              MPI_DOUBLE, 0, comm);
19        free(a);
20     } else {
21        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22              MPI_DOUBLE, 0, comm);
23     }
24  }  /* 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  */);
```

MPI_Gather function signature

MPI_Gather Illustration

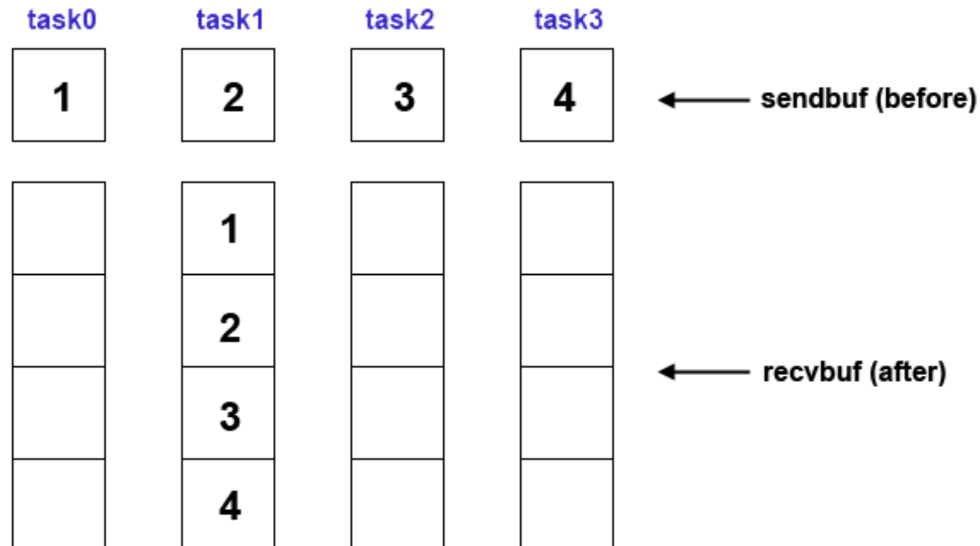


image: LLNL MPI tutorial

Print a Distributed Vector

```
1   void Print_vector(
2         double    local_b[]  /* in */,
3         int       local_n    /* in */,
4         int       n          /* in */,
5         char      title[]    /* in */,
6         int       my_rank    /* in */,
7         MPI_Comm  comm       /* in */) {
8
9      double* b = NULL;
10     int i;
11
12     if (my_rank == 0) {
13        b = malloc(n*sizeof(double));
14        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15              MPI_DOUBLE, 0, comm);
16        printf("%s\n", title);
17        for (i = 0; i < n; i++)
18           printf("%f ", b[i]);
19        printf("\n");
20        free(b);
21     } else {
22        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23              MPI_DOUBLE, 0, comm);
24     }
25  }  /* 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  */);
```

MPI_Allgather function signature

MPI_Allgather Illustration

MPI_Allgather

Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```

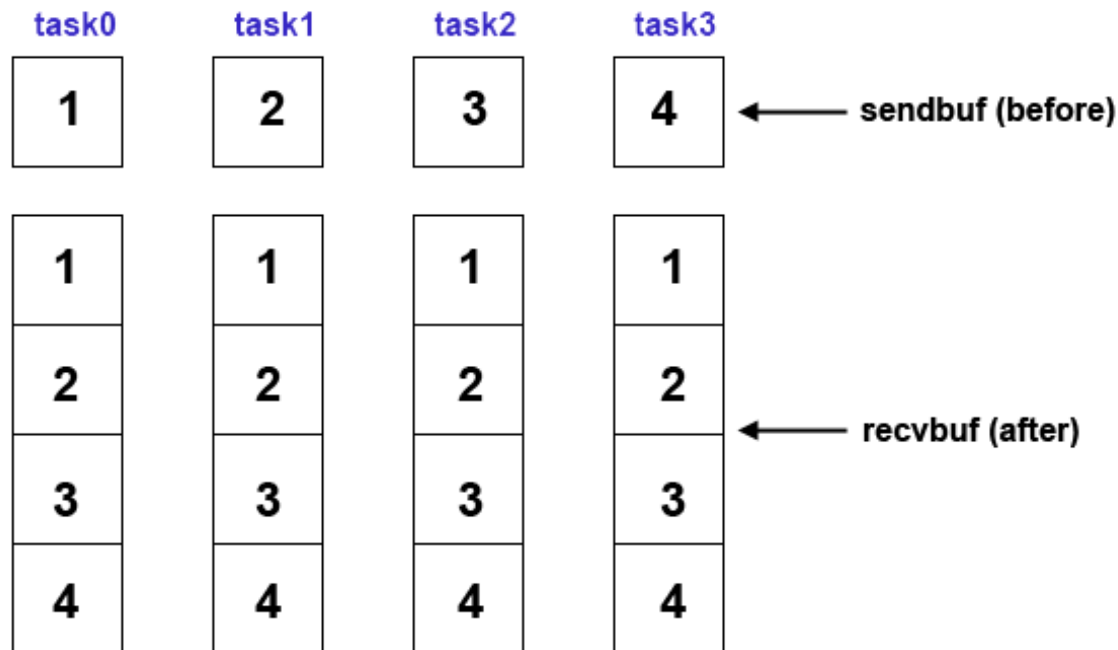


image: LLNL MPI tutorial

Example: Vector Addition

- **See source code**
  - vector_add.c (serial code)
  - mpi_vector_add.c (MPI code)

Use Case: 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}

Figure 3.11

```
/* 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 for Matrix-Vector Multiplication

Use Case: Matrix-Vector Multiplication (2)

- **The C programming langue**
  - tedious idiosyncrasies with handling multi-dimensional arrays
  - not possible to write generic functions that accept multi-dimensional arrays of varying size
- **Hence, programmers typically express two (or higher)-dimensional arrays as one-dimensional arrays (linearized storage)**

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

stored as $0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11.$

- **Requires explicit mapping from higher-dimensions to single dimension in the code**

Use Case: Matrix-Vector Multiplication (3)

```
1   void Mat_vect_mult(
2         double  A[]  /* in  */,
3         double  x[]  /* in  */,
4         double  y[]  /* out */,
5         int     m    /* in  */,
6         int     n    /* in  */) {
7      int i, j;
8
9      for (i = 0; i < m; i++) {
10        y[i] = 0.0;
11        for (j = 0; j < n; j++)
12           y[i] += A[i*n+j]*x[j];
13     }
14  }  /* Mat_vect_mult */
```

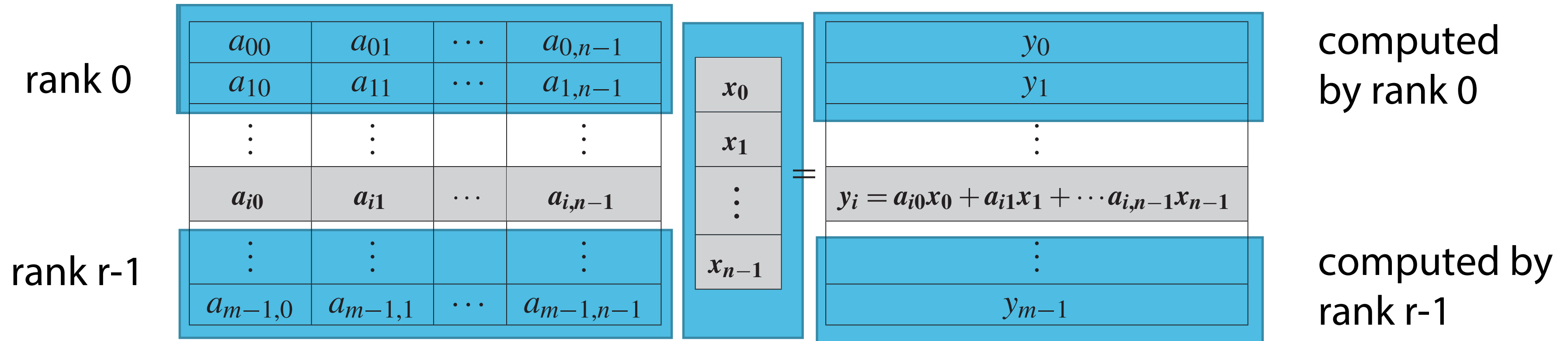
A[],x[],y[] are linear 1D arrays

matrix dimensions explicitly passed to function

explicit mapping from higher dimensions to 1D array index

Serial Pseudo-Code for Matrix-Vector Multiplication with linearized arrays

Use Case: Matrix-Vector Multiplication (4)



rows of matrix A will be block-distributed to all ranks

x will be copied to all ranks

Example: Matrix-Vector Multiplication

- **See source code**
  - mat_vect_mult.c (serial code)
  - mpi_mat_vect_mult.c (MPI code)

MPI matrix-vector multiplication function

```
1   void Mat_vect_mult(
2         double    local_A[]  /* in   */,
3         double    local_x[]  /* in   */,
4         double    local_y[]  /* out  */,
5         int       local_m    /* in   */,
6         int       n          /* in   */,
7         int       local_n    /* in   */,
8         MPI_Comm  comm       /* in   */) {
9      double* x;
10     int local_i, j;
11     int local_ok = 1;
12
13     x = malloc(n*sizeof(double));
14     MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15           x, local_n, MPI_DOUBLE, comm);
16
17     for (local_i = 0; local_i < local_m; local_i++) {
18        local_y[local_i] = 0.0;
19        for (j = 0; j < n; j++)
20           local_y[local_i] += local_A[local_i*n+j]*x[j];
21     }
22     free(x);
23  }  /* Mat_vect_mult */
```

code shows only distribution of vector x (but not A)

MPI 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**
- **Allow efficient data handling**
  - function that sends data knows this information about a collection of data items, allowing it to collect the items from memory before they are sent
  - function that receives data can distribute the items into their correct destinations in memory when they're received
- **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

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

Creating Derived Datatypes

- **Build 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 Using 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};

   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 Using Derived Datatype (2)

```
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**
  - MPI_Wtime available as part of MPI library

```
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**
- **"timer.h" may not be part of default include path**
    - you may have to inform the compiler about the directory containing timer.h
    - in gcc add option: -I/path/to/timer_h

```
#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 function signature

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

	Order of Matrix				
comm_sz	*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

(milliseconds)

Reminder: Speedup and Efficiency

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

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

Speedups and Efficiency of Parallel Matrix-Vector Multiplication

speedup

	Order of Matrix				
comm_sz	*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

efficiency

	Order of Matrix				
comm_sz	*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**
  - strongly scalable: programs can maintain a constant efficiency without increasing the problem size
  - weakly scalable: program maintain a constant efficiency if the problem size increases at the same rate as the number of processes

Parallelizing a Sorting Algorithm

- **n keys and p = 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

```
1   void Bubble_sort(
2           int  a[]  /* in/out */,
3           int  n    /* in     */) {
4       int list_length, i, temp;
5
6       for (list_length = n; list_length >= 2; list_length--)
7           for (i = 0; i < list_length-1; i++)
8               if (a[i] > a[i+1]) {
9                   temp = a[i];
10                  a[i] = a[i+1];
11                  a[i+1] = temp;
12              }
13
14  }   /* 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)
  - result: 5, 9, 3, 4
- **Odd phase**
  - compare-swap (9,3)
  - result: 5, 3, 9, 4
- **Even phase**
  - compare-swap (5,3) and (9,4)
  - result: 3, 5, 4, 9
- **Odd phase**
  - compare-swap (5,4)
  - result: 3, 4, 5, 9

Serial Odd-Even Transposition Sort

```
1   void Odd_even_sort(
2         int  a[]  /* in/out */,
3         int  n    /* in     */) {
4      int phase, i, temp;
5
6      for (phase = 0; phase < n; phase++)
7         if (phase % 2 == 0) { /* Even phase */
8            for (i = 1; i < n; i += 2)
9               if (a[i-1] > a[i]) {
10                 temp = a[i];
11                 a[i] = a[i-1];
12                 a[i-1] = temp;
13              }
14        } else { /* Odd phase */
15           for (i = 1; i < n-1; i += 2)
16              if (a[i] > a[i+1]) {
17                 temp = a[i];
18                 a[i] = a[i+1];
19                 a[i+1] = temp;
20              }
21        }
22  }  /* Odd_even_sort */
```

Communications Among Tasks in Odd-Even Sort

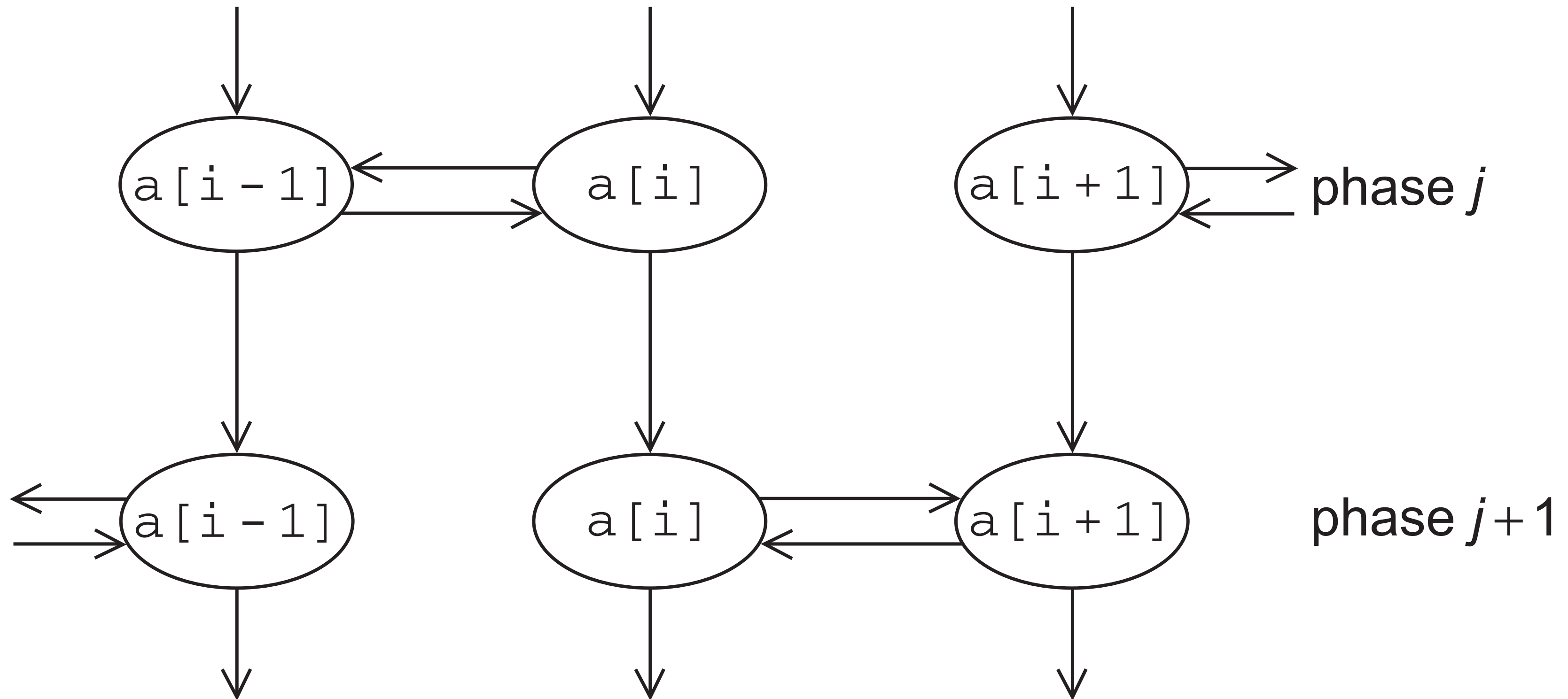


Figure 3.11

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

Theorem: If parallel odd-even transposition sort is run with p processes, then after p phases the list will be sorted.

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;
```

MPI_PROC_NULL is a constant defined by MPI. If used as source or destination, no communication will take place and the communication call will simply return

Watch Out for Deadlocks

- **A straight-forward implementation would communicate as follows**

```
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).
```

- **Problem**

MPI rank 1

`MPI_Send(… dest=2, ...)`

may block until received by rank 2

MPI rank 2

`MPI_Send(… dest=1, ...)`

may block until received by rank 1

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
- **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
- **Hence, if MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv**
  - 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.**

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 */);
```

MPI_Ssend function signature

- **We can experimentally check whether a program is safe by replacing all MPI_Send calls with MPI_Ssend and check for crashes or deadlocks**

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).
```

Manual communication scheduling:
change order of send and receive for every other process

```
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

0 1 2 4 3

Time 2

```
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   */);
```

MPI_Sendrecv function signature

MPI_Sendrecv_replace

```
int MPI_Sendrecv_replace(
      void*         buf_p          /* in/out */,
      int           buf_size       /* in     */,
      MPI_Datatype  buf_type       /* in     */,
      int           dest           /* in     */,
      int           send_tag       /* in     */,
      int           source         /* in     */,
      int           recv_tag       /* in     */,
      MPI_Comm      communicator   /* in     */,
      MPI_Status*   status_p       /* in     */);
```

MPI_Sendrecv_replace function signature

same functionality as MPI_Sendrecv but replaces data in send buffer with received data

Parallel Odd-Even Transposition Sort

- **Further optimization, sort local lists only once**

```
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 */
```

merging avoids sorting lists that are already sorted

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
- 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 (strongly) scalable
- An MPI program is unsafe if its correct behavior depends on the fact that MPI_Send is buffering its input

Acknowledgements

- **Peter S. Pacheco / Elsevier**
  - for providing the lecture slides on which this presentation is based
- **Illustrations for MPI collectives taken from the excellent MPI tutorial published by Lawrence Livermore National Lab**
  - https://computing.llnl.gov/tutorials/mpi/

Change log

- **1.2.3 (2017-11-20)**
  - add slide 102 to clarify deadlock situation
  - clarify slides 108, 109
- **1.2.2 (2017-11-06)**
  - clarify slides 49, 72
  - remove slide 102 (safe communication with 5 processes)
- **1.2.1 (2017-10-24)**
  - extend table of contents (teaser for Advanced MPI section)
  - clarify slide 20, 22
  - fix error on slide 38 (also in book) Get_input instead of Get_data
  - correction of spelling and grammatical errors
- **1.2.0 (2017-10-23)**
  - updated for winter term 2017/18
  - clarify slide 16
  - add information on wildcards slide 21
  - cosmetics (title case)

Change log

- **1.1.0 (2017-07-13)**
  - fix typos on slide 40
- **1.0.3 (2016-12-01)**
  - add slide 71 for clarification of matrix / vector distribution
  - cosmetics and correction of minor typos
- **1.0.2 (2016-11-29)**
  - slide 49, clarify kind of reduction and receiver in example
  - add slide numbers everywhere
- **1.0.1 (2016-11-24)**
  - cosmetic changes to part 1, initial version of remaining slides
- **1.0.0 (2016-11-24)**
  - initial version of slides (part 1)