COMP/CS 605: Introduction to Parallel Computing
Topic : MPI:: Point to Point and Collective Communications

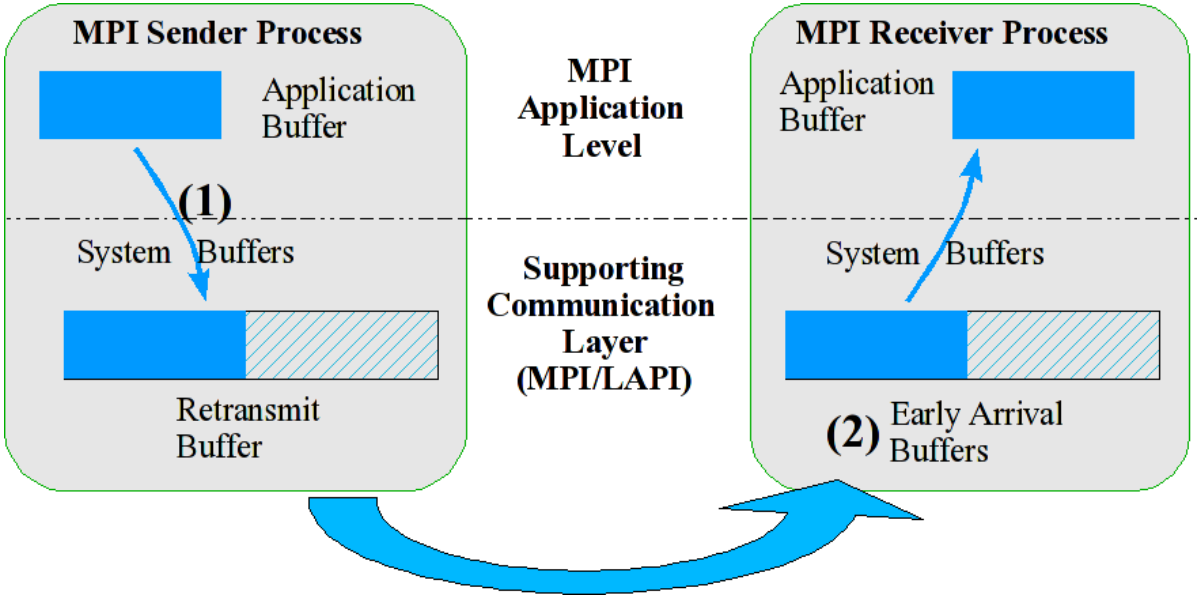
Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Presented: 02/16/17
Updated: 02/16/17

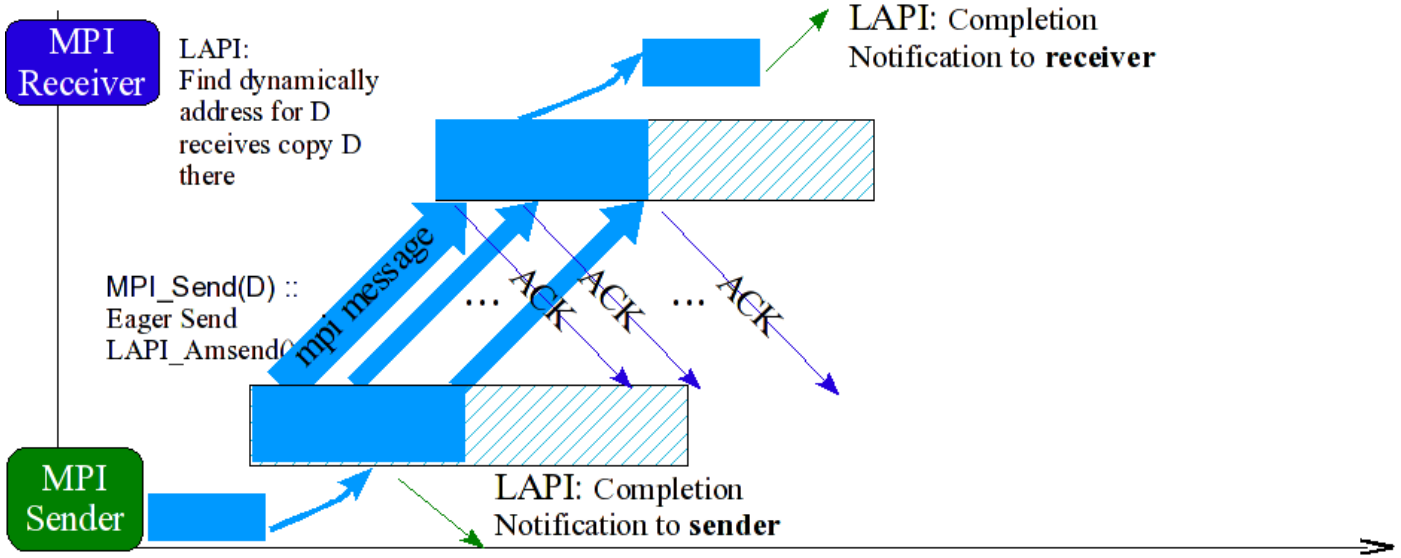
Table of Contents

MPI: Point-to-Point Communications



- **Point-to-Point:** uses **MPI_SEND** and **MPI_RECV**.
- Two PEs transfer data from one to the other *only*.
- Blocking: PE #1 posts a **SEND** operation
- Target (PE#2) process posts a **RECEIVE** for data being transferred.

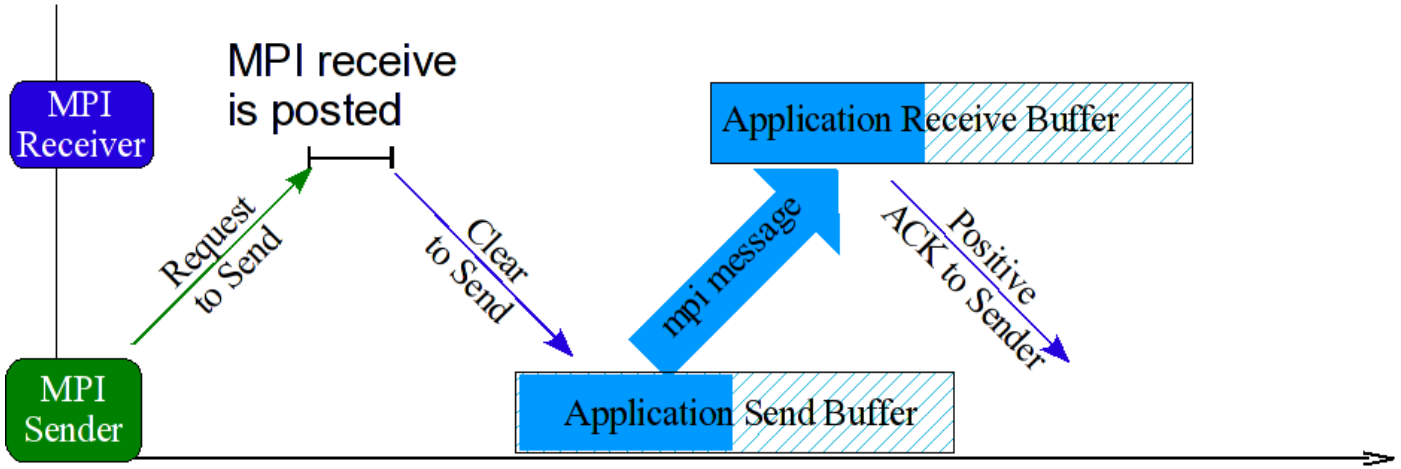
Image Source: `http://sc.tamu.edu/systems/hydra/hardware.php`
Note: LAPI is the IBM Low-level Application Programming Interface



- Smaller messages are send immediately.
- "Early Arrival" buffer: for messages where RECV has not been posted
- User can [optionally] set env vars:
  - MP_EAGER_LIMIT: maximum message size (default = 32768 bytes)
  - MP_BUFFER_MEM: amount allowed on receiver end.

Image Source: `http://sc.tamu.edu/systems/hydra/hardware.php`

MPI: Message Buffering - Rendevous Protocol



- For larger messages.
- sender notifies receiver that there is data to be sent.
- Receiver responds when a receive is posted.
- Sender transmits actual message after it receives acknowledgment (ACK).
- Data is saved directly to application receive buffer

Image Source: `http://sc.tamu.edu/systems/hydra/hardware.php`

MPI: Messages

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status))

- What is the message being sent?
- MPI_Send/Recv message contains the data and specific information about:
  - source task/processor
  - destination task/processor
  - tag for identification of message
  - communicator

Blocking MPI_Send

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

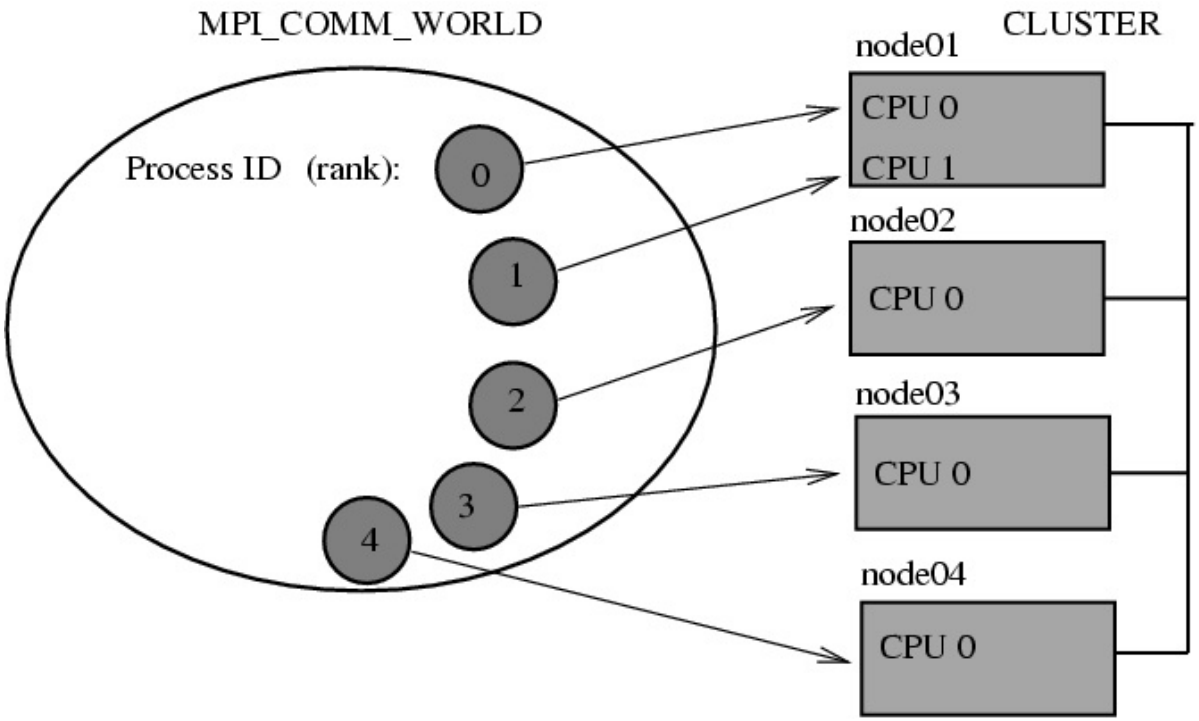
MPI_Send (		
buf	x	Initial address of send buffer (choice)
count	::	number of elements in send buffer (nonnegative integer)
datatype	::	datatype of each send buffer element (handle)
dest	::	rank of destination (integer)
tag	::	message tag (integer)
comm	::	communicator (handle)
)		

Modify mpi_hello.c to do a simple operation (mpi_add.c)

```
if (my_rank != ROOT) {
    /* Create message */
    printf("Greetings from Worker: %d of %d!\n", my_rank, comm_sz);

    /* Send message to  ROOT */
    MPI_Send(&my_rank, 1, MPI_INT, ROOT, tag, MPI_COMM_WORLD);

} else {
    sum=0;
    /* Print my message */
    printf("Greetings from Master: %d of %d!\n", my_rank, comm_sz);

    for (int q = 1; q < comm_sz; q++) {

        /* Receive message from process q */
        MPI_Recv(&p, 1, MPI_INT, q, tag, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

        /* Print message from process q */
        printf("P[%d]: RECV Data From Worker node[%d]\n",my_rank, p);
        sum=sum+p;
    }
}
printf("P[%d]: SUM of data=%d\n",my_rank, sum);
/* Shut down MPI */
MPI_Finalize();
```

Output from mpi_add.c

```
[mpi-hello]mthomas%  mpicc -o mpi_add mpi_add.c

[mpi-hello]mthomas% mpirun -np 4 ./mpi_add

Greetings from Master: 0 of 4!
Greetings from Worker: 3 of 4!
P[3]: SUM of data=0
Data From Proc: 1
Data From Proc: 2
Data From Proc: 3
Greetings from Worker: 1 of 4!
P[1]: SUM of data=0
Greetings from Worker: 2 of 4!
P[2]: SUM of data=0
P[0]: SUM of data=6
```



Communication on a multimode cluster with multiple PE's

Program That Might Hang

```
printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
if (my_rank == 0) {
    printf("P[%d] Ready to RECV Data From %d nodes.\n", my_rank, comm_sz)
    for (int q = 1; q < comm_sz; q++) {
        MPI_Recv(&p, 1, MPI_INT, q, tag, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Data From Proc: %d\n", p);
        sum=sum+p;
        printf("P[%d] Ready to SEND Data to Node: %d.\n", my_rank, q);
        MPI_Send(&q, 1, MPI_INT, q, tag, MPI_COMM_WORLD);
    }
} else {
    printf("P[%d] Ready to RECV data from Proc %d.\n", my_rank, ROOT);
    MPI_Recv(&p, 1, MPI_INT, ROOT, tag, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    /* Create message */
    printf("P[%d] Ready to SEND my data to: %d.\n", my_rank, ROOT);
    MPI_Send(&my_rank, 1, MPI_INT, ROOT, tag, MPI_COMM_WORLD);
}
printf("P[%d]: SUM of data=%d\n",my_rank, sum);
/* Shut down MPI */
MPI_Finalize();
```

Output from mpi_add_hang.c: NP=4 runs fine

```
[mpi-hello] mthomas% mpicc -o mpi_add_hang mpi_add_hang.c
[mpi-hello] mthomas% mpirun -np 4 ./mpi_add_hang
[mpi-hello] mthomas% mpicc -o mpi_add_hang mpi_add_hang.c
[mpi-hello]  mthomas% mpirun -np 4 ./mpi_add_hang
Greetings from process 0 of 4!
P[0] Ready to RECV Data From 4 nodes.
Greetings from process 1 of 4!
P[1] Ready to RECV data from Proc 0.
Greetings from process 2 of 4!
P[2] Ready to RECV data from Proc 0.
Greetings from process 3 of 4!
P[3] Ready to RECV data from Proc 0.
^CCtrl-C caught... cleaning up processes
```

Output from mpi_add_hang.c: NP=16 hangs

```
[gidget mthomas]% mpirun -np 16 ./mpi_add_hang
Greetings from process 1 of 16!
P[1] Ready to RECV data from Proc 0.
Greetings from process 4 of 16!
P[4] Ready to RECV data from Proc 0.
Greetings from process 6 of 16!
P[6] Ready to RECV data from Proc 0.
Greetings from process 8 of 16!
P[8] Ready to RECV data from Proc 0.
Greetings from process 10 of 16!
P[10] Ready to RECV data from Proc 0.
Greetings from process 12 of 16!
P[12] Ready to RECV data from Proc 0.
Greetings from process 14 of 16!
P[14] Ready to RECV data from Proc 0.
Greetings from process 15 of 16!
P[15] Ready to RECV data from Proc 0.
Greetings from process 2 of 16!
P[2] Ready to RECV data from Proc 0.
Greetings from process 9 of 16!
P[9] Ready to RECV data from Proc 0.
Greetings from process 11 of 16!
P[11] Ready to RECV data from Proc 0.
Greetings from process 3 of 16!
P[3] Ready to RECV data from Proc 0.
Greetings from process 7 of 16!
P[7] Ready to RECV data from Proc 0.
Greetings from process 13 of 16!
P[13] Ready to RECV data from Proc 0.
Greetings from process 0 of 16!
P[0] Ready to RECV Data From 16 nodes.
Greetings from process 5 of 16!
P[5] Ready to RECV data from Proc 0.
```

MPI: Point-to-Point Communications

Terminal — top — 127×41

```
Processes: 195 total, 18 running, 2 stuck, 175 sleeping, 1060 threads
Load Avg: 13.60, 5.06, 2.20  CPU usage: 75.74% user, 24.25% sys, 0.0% idle  SharedLibs: 17M resident, 5944K dat
MemRegions: 43171 total, 3532M resident, 147M private, 2430M shared.
PhysMem: 2283M wired, 6184M active, 1291M inactive, 9758M used, 6617M free.
VM: 423G vsize, 1053M framework vsize, 6652096(0) pageins, 0(0) pageouts. Networks: packets: 871787/798M in, 66
Disks: 674727/8487M read, 545880/11G written.

PID   COMMAND      %CPU TIME     #TH  #WQ  #POR #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE  PGRP PPID STATE    UID
6130  Grab         4.9  00:00.49 4    3    161+ 149+  5496K+ 46M+   15M+   84M+   2530M+ 6130 154  sleeping 501
6127  mdworker     0.0  00:01.07 2    1    56   87    5992K  18M    11M    66M    2471M  6127 149  sleeping 89
6126  mdworker     0.0  00:01.26 2    1    53   86    5620K  19M    11M    57M    2469M  6126 154  sleeping 501
6125  top          23.0 00:06.08 1/1  0    27   30    1848K  216K   2592K  17M    2376M  6125 6079 running  0
6124  mdworker     0.0  00:00.07 3    1    56   71    1232K  13M    5832K  53M    2424M  6124 154  sleeping 501
6123  pdftex       0.0  00:03.36 1    0    18   57    39M    1656K  40M    117M   2485M  6123 5751 sleeping 501
6122  mpi_add_hang 51.8 00:45.83 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6121  mpi_add_hang 47.9 00:46.31 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6120  mpi_add_hang 49.4 00:45.94 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6119  mpi_add_hang 47.6 00:46.07 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6118  mpi_add_hang 45.8 00:45.78 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6117  mpi_add_hang 44.9 00:45.70 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6116  mpi_add_hang 46.3 00:46.18 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6115  mpi_add_hang 47.0 00:46.08 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6114  mpi_add_hang 51.2 00:46.07 1/1  0    18   36    324K   31M    1568K  18M    2457M  6105 6106 running  501
6113  mpi_add_hang 47.4 00:46.10 1/1  0    18   36    324K   31M    1568K  18M    2457M  6105 6106 running  501
6112  mpi_add_hang 46.7 00:46.11 1/1  0    18   36    324K   31M    1568K  18M    2457M  6105 6106 running  501
6111  mpi_add_hang 50.7 00:45.56 1/1  0    18   36    324K   31M    1568K  18M    2457M  6105 6106 running  501
6110  mpi_add_hang 49.4 00:46.15 1/1  0    18   36    324K   31M    1568K  18M    2457M  6105 6106 running  501
6109  mpi_add_hang 46.7 00:46.26 1/1  0    18   36    324K   31M    1568K  18M    2457M  6105 6106 running  501
6108  mpi_add_hang 44.7 00:46.10 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6107  mpi_add_hang 46.0 00:46.07 1/1  0    18   36    324K   31M    1564K  18M    2457M  6105 6106 running  501
6106  hydra_pmi_pr 0.0  00:00.00 1    0    17   27    204K   224K   712K   9768K  2376M  6105 6105 sleeping 501
6105  mpiexec.hydr 0.0  00:00.00 1    0    18   28    232K   228K   828K   9644K  2376M  6105 6084 sleeping 501
```

Blocking MPI_SendRecv

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
int dest, int sendtag,
void *recvbuf, int recvcount, MPI_Datatype recvtype,
int source, int recvtag,
MPI_Comm comm, MPI_Status *status)

MPI_SendRecv (

sendbuf	::	Initial address of send buffer (choice)
sendcount	::	number of elements in send buffer (nonnegative integer)
sendtype	::	datatype of each send buffer element (handle)
dest	::	rank of destination (integer)
sendtag	::	message tag (integer)
recvbuf	::	Initial address of send buffer (choice)
recvcount	::	number of elements in send buffer (nonnegative integer)
recvtype	::	datatype of each send buffer element (handle)
source	::	rank of destination (integer)
recvtag	::	message tag (integer)
comm	::	communicator (handle)

)

Code: mpi_add_sendrecv.c

```
!--Exchange messages
      tag1=1;  tag2=2
    printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
    if (my_rank == 0) {
       sum=0;
       for (int q = 1; q < comm_sz; q++) {
          printf("P[%d] Master node ready for SendRecv Op with
                          Worker[%d]  \n",  my_rank,q);
          sbuf=q*q;
          MPI_Sendrecv(&sbuf,1,MPI_INT,q,tag1,
                       &p,    1,MPI_INT,q,tag2,MPI_COMM_WORLD,
                       &status);
          sum=sum+p;
       }
    } else {
       printf("P[%d] Worker node ready for SendRecv Op with Master[%d].\n",
                              my_rank,ROOT);
          MPI_Sendrecv(&my_rank,1,MPI_INT,ROOT,tag2,
                       &rbuf,    1,MPI_INT,ROOT,tag1,MPI_COMM_WORLD,
                       &status);
    }
    printf("P[%d]: My rbuf=[%d], my sum=[%d]\n",my_rank,rbuf,sum);
    MPI_Finalize();        /* Shut down MPI */
```

```
/* listing:  mpi_add.c  */
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */
const int MAX_STRING = 100;
int main(void) {
   char       greeting[MAX_STRING];  /* String storing message */
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int    p, sum, tag=1, ROOT=0;
   MPI_Init(NULL, NULL);                          /* Start up MPI */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);     /* Get the number of processes */
   /* Get my rank among all the processes */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if (my_rank != ROOT) {
      printf("Greetings from Worker: %d of %d!\n", my_rank, comm_sz);
      /* Send message to  ROOT */
      MPI_Send(&my_rank, 1, MPI_INT, ROOT, tag, MPI_COMM_WORLD);
   } else {
      /* Print my message */
      printf("Greetings from Master: %d of %d!\n", my_rank, comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         /* Receive message from process q */
         MPI_Recv(&p, 1, MPI_INT, q, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("P[%d]: RECV Data From Worker node[%d]\n",my_rank, p);
         sum=sum+p;
      }
   }
   printf("P[%d]: SUM of data=%d\n",my_rank, sum);
   /* Shut down MPI */
   MPI_Finalize();
}
```

```
/*  FULL CODE LISTING (slide 1/2)
 * File:
 *     mpi_add_sendrecv.c
 *  modified by MThomas based on mpi_hello.c
 * Purpose:
 *     A "add,world" program that uses MPI
 *
 * Compile:
 *     mpicc -g -Wall -std=C99 -o mpi_add mpi_add.c
 */

#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */
const int MAX_STRING = 100;
int main(void) {
   char        greeting[MAX_STRING];  /* String storing message */
   int         comm_sz;               /* Number of processes    */
   int         my_rank;               /* My process rank        */
   int         p,q, rbuf,sbuf, sum, ROOT=0;
   int         tag1=1, tag2=2;
   MPI_Status   status;

      /* Start up MPI */
   MPI_Init(NULL, NULL);

   /* Get the number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   /* Get my rank among all the processes */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

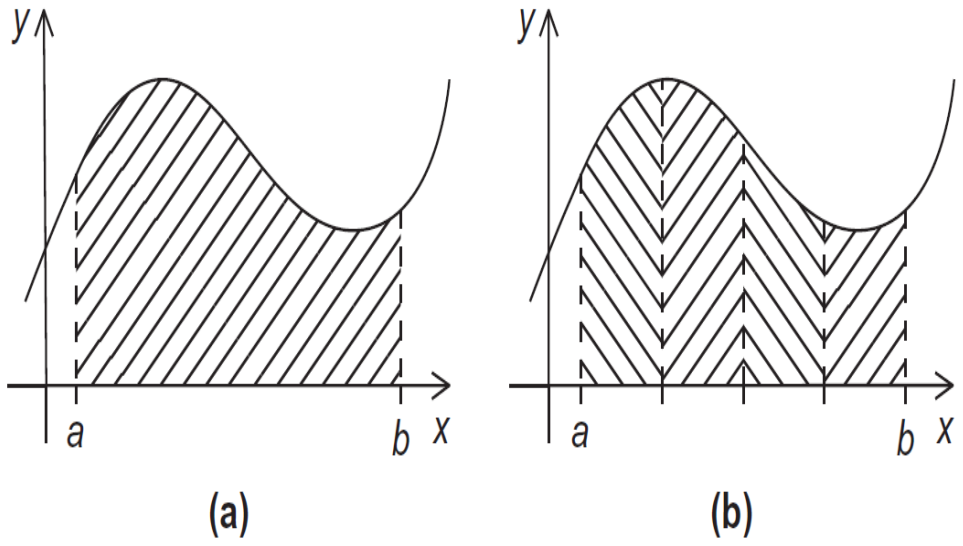
```
    printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
    if (my_rank == 0) {
       sum=0;
       for (int q = 1; q < comm_sz; q++) {
          printf("P[%d] Master node ready for SendRecv Op with Worker[%d]  \n", my_rank,q);
          sbuf=q*q;
       /*
       */
          MPI_Sendrecv(&sbuf,1,MPI_INT,q,tag1,
                       &p,   1,MPI_INT,q,tag2,MPI_COMM_WORLD,
                       &status);
            sum=sum+p;
       }
    } else {
       printf("P[%d] Worker node ready for SendRecv Op with Master[%d].\n", my_rank,ROOT);
       /*
       */
          MPI_Sendrecv(&my_rank,1,MPI_INT,ROOT,tag2,
                       &rbuf,   1,MPI_INT,ROOT,tag1,MPI_COMM_WORLD,
                       &status);
    }
    printf("P[%d]: My rbuf=[%d], my sum=[%d]\n",my_rank,rbuf,sum);
    /* Shut down MPI */
    MPI_Finalize();

    return 0;
}  /* main */
```

Output from mpi_add_sendrecv.c

```
[mpi-hello]mthomas% mpicc -o mpi_add_hang mpi_add_hang.c
[mpi-hello]mthomas% mpirun -np 4 ./mpi_add_hang
[mpi-hello]mthomas% mpirun -np 4 ./mpi_add_sendrecv | sort | grep "Master n
P[0] Master node ready for SendRecv Op with Worker[1]
P[0] Master node ready for SendRecv Op with Worker[2]
P[0] Master node ready for SendRecv Op with Worker[3]
[mpi-hello]mthomas% mpirun -np 4 ./mpi_add_sendrecv | sort | grep "Worker n
P[1] Worker node ready for SendRecv Op with Master[0].
P[2] Worker node ready for SendRecv Op with Master[0].
P[3] Worker node ready for SendRecv Op with Master[0].
[mpi-hello]mthomas% mpirun -np 4 ./mpi_add_sendrecv | sort | grep rbuf
P[0]: My rbuf=[0], my sum=[6]
P[1]: My rbuf=[1], my sum=[0]
P[2]: My rbuf=[4], my sum=[0]
P[3]: My rbuf=[9], my sum=[0]
```

Recall: The Trapezoid Rule for Numerical Integration

Solve the Integral: $\int_a^b F(x)dx$

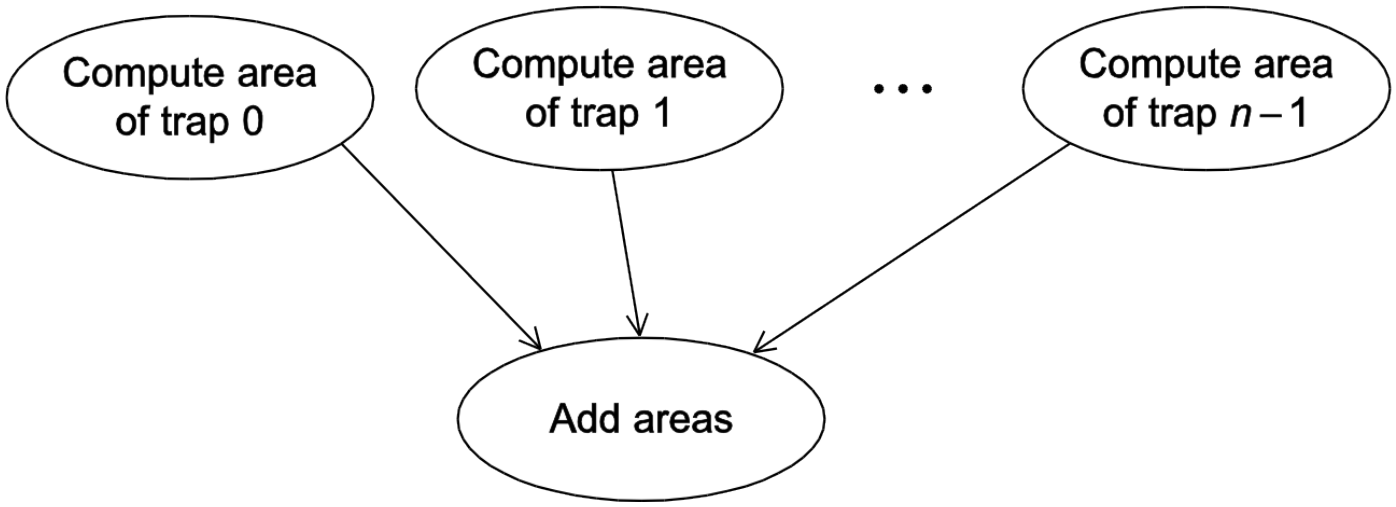


Where $F(x)$ can be any function of x: $f\left(x^2\right)$, $f\left(x^3\right)$ See Pacheco (2011), Ch3.

Parallelizing the Trapezoidal Rule

- Partition problem solution into tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores

Tasks and communications for Trapezoidal Rule



Master Node collects sums using MPI_SEND / MPI_RECV
$\vartheta\,(np-1)$**, where** np **is the number of** PEs

Output shows 7 Send/Recv pairs: $\vartheta(np-1) = \vartheta(7)$, for $np = 8$

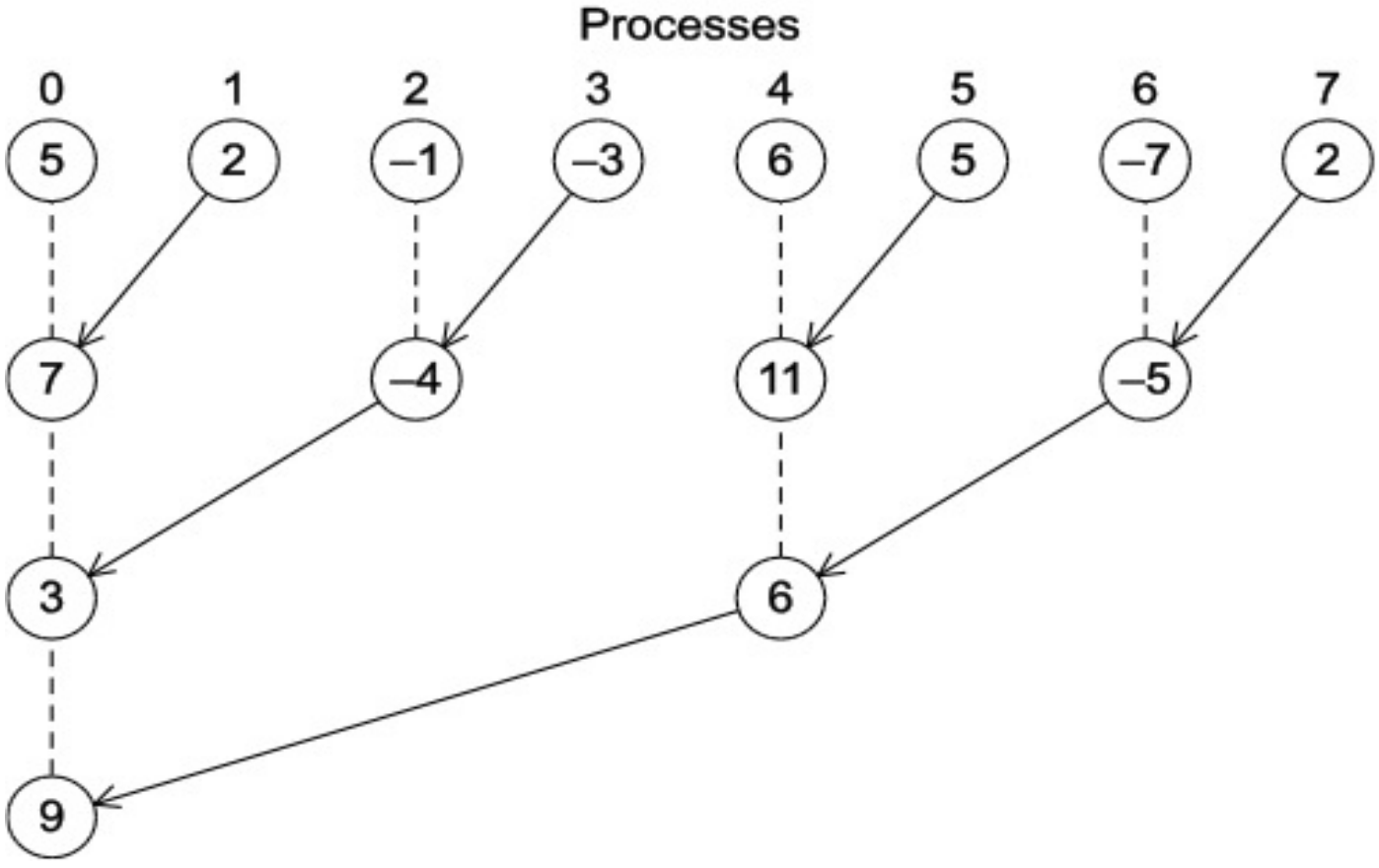
```
[mthomas]% mpirun -np 8 ./mpi_trap2
Enter a, b, and n
1 50 100
PE[0] RECV from PE[1]: local estimate=584.199266
PE[0] RECV from PE[2]: local estimate=1466.537954
PE[0] RECV from PE[3]: local estimate=2755.471586
PE[0] RECV from PE[4]: local estimate=4451.000162
PE[0] RECV from PE[5]: local estimate=6553.123682
PE[0] RECV from PE[6]: local estimate=9061.842146
PE[0] RECV from PE[7]: local estimate=11977.155554
PE[1] SEND: For 12 trapezoids, local estimate=584.199266
PE[2] SEND: For 12 trapezoids, local estimate=1466.537954
PE[3] SEND: For 12 trapezoids, local estimate=2755.471586
PE[4] SEND: For 12 trapezoids, local estimate=4451.000162
PE[5] SEND: For 12 trapezoids, local estimate=6553.123682
PE[6] SEND: For 12 trapezoids, local estimate=9061.842146
PE[7] SEND: For 12 trapezoids, local estimate=11977.155554
With n = 100 trapezoids, our estimate
of the integral from 1.000000 to 50.000000 = 3.695778587200000e+04
```

Tree-structured communication

Reduces the Number of Communications:

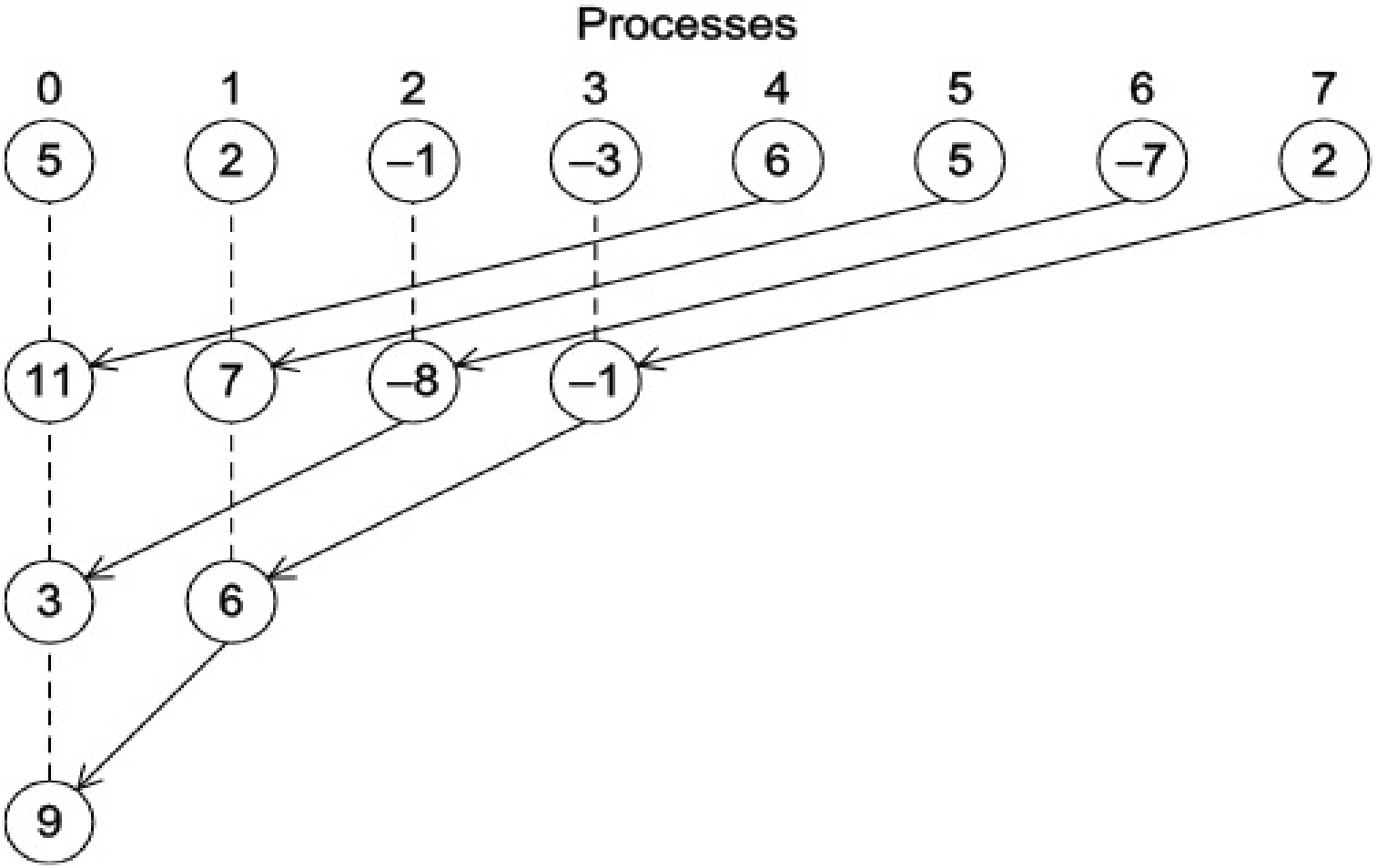
1. In the first phase:
   (a) Process 1 sends to 0; 3 sends to 2; 5 sends to 4; and 7 sends to 6.
2. Next phase:
   (a) Processes 0, 2, 4, and 6 add in the received values.
   (b) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (c) Processes 0 and 4 add the received values into their new values
3. Final phase:
   (a) Process 4 sends its newest value to process 0.
   (b) Process 0 adds the received value to its newest value.

Tree-structured Global Sum V1



Communication pattern used by 1st version of trap.c algorithm:
7 messages, 7 adds by node 0.
With tree-structured Comm: master has 3 messages and 3 adds.

Tree-structured Global Sum V2.



Collective Communication: 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);
```

Operator passed as an argument. Count > 1 supports arrays

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 communicator processes must call same collective function.
  - e.g.: program attempts to match a call to MPI_Reduce on PE(i) with a call to MPI_Recv on PE(j) will cause program will hang or crash.
- Arguments passed by each process to an MPI collective communication must be compatible.
  - e.g.: PE(i) passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce causes code to hang or crash.
- The output_data_p argument is only used on dest_process.
- All processes need to pass argument corresponding to output_data_p
- Point-to-point communications are matched on the basis of tags and communicators.
- Collective communications matched by communicator and order called

Example: What happens when we have multiple calls to MPI_Reduce?

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, ...)`

$tinit$: $p0[a=0, b=0, c=0, d=0]$; $p1[a=0, b=0, c=0, d=0]$; $p2[a=0, b=0, c=0, d=0]$
$t0$: $p0[a=1, b=0, c=2, d=0]$; $p1[a=1, b=0, c=2, d=0]$; $p2[a=1, b=0, c=2, d=0]$
$t1$: $p0[a=1, b=1, c=2, d=0]$; $p1[a=1, b=0, c=2, d=2]$; $p2[a=1, b=1, c=2, d=0]$
$on\ p0$: $b = P0(a) + P1(c) + P2(a) = 1 + 2 + 1 = 4,\ d = 0$
$t2$: $p0[a=1, b=2, c=2, d=2]$; $p1[a=1, b=1, c=2, d=0]$; $p2[a=1, b=0, c=2, d=2]$
$on\ p0$: $d = P0(c) + P1(a) + P2(c) = c + a + c = 2 + 1 + 2 = 5$

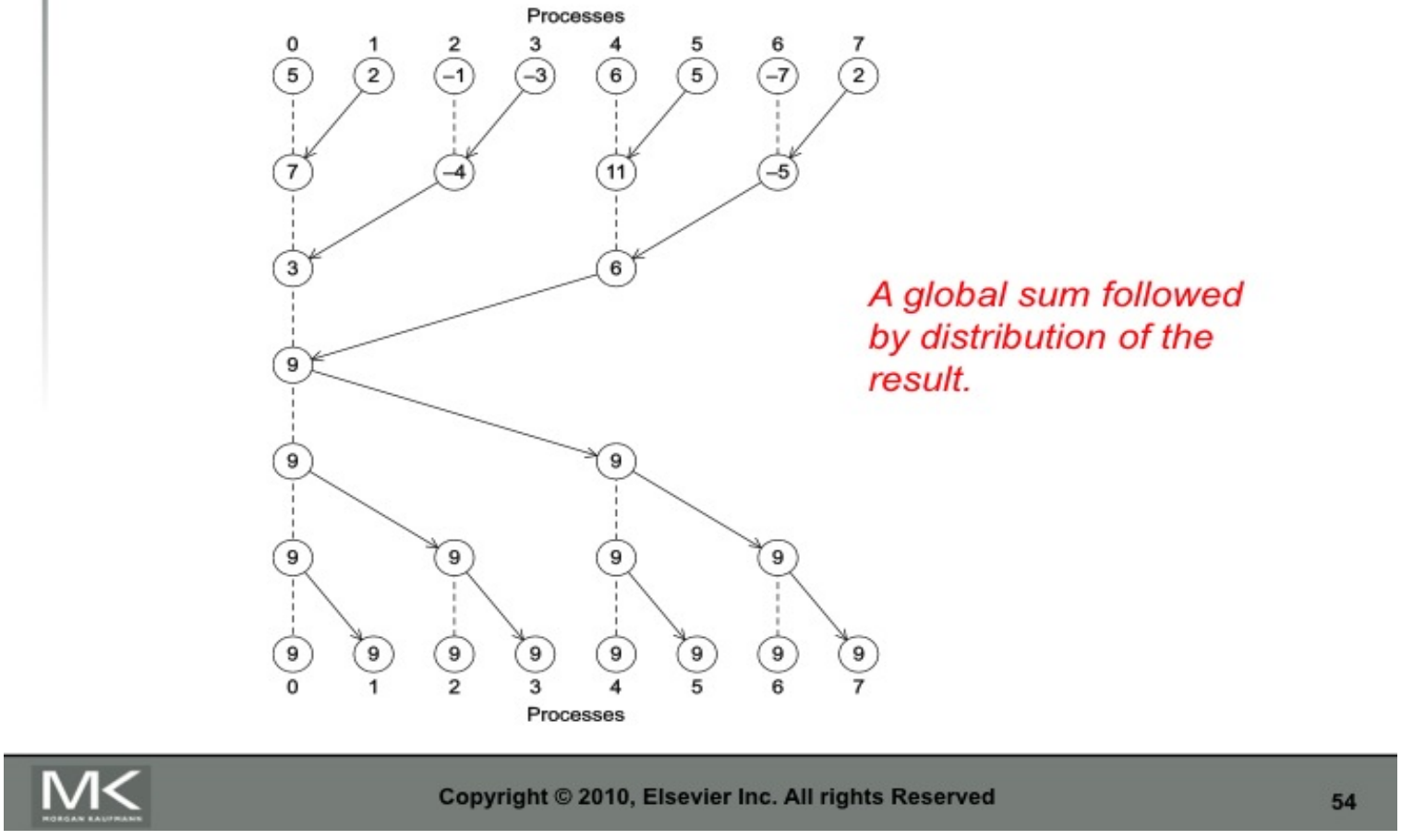
Collective vs. Point-to-Point Communications

- 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.
- 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 $2+1+2 = 5$.

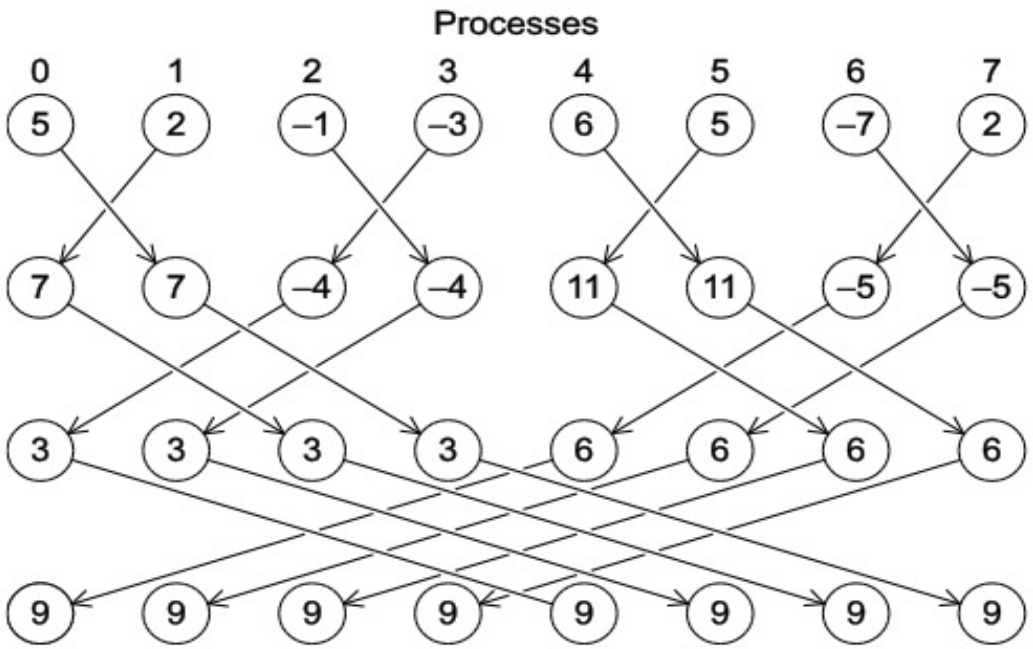
MPI_Allreduce

- 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  */);
```



MPI_AllReduce: Butterfly communication pattern, O(n)



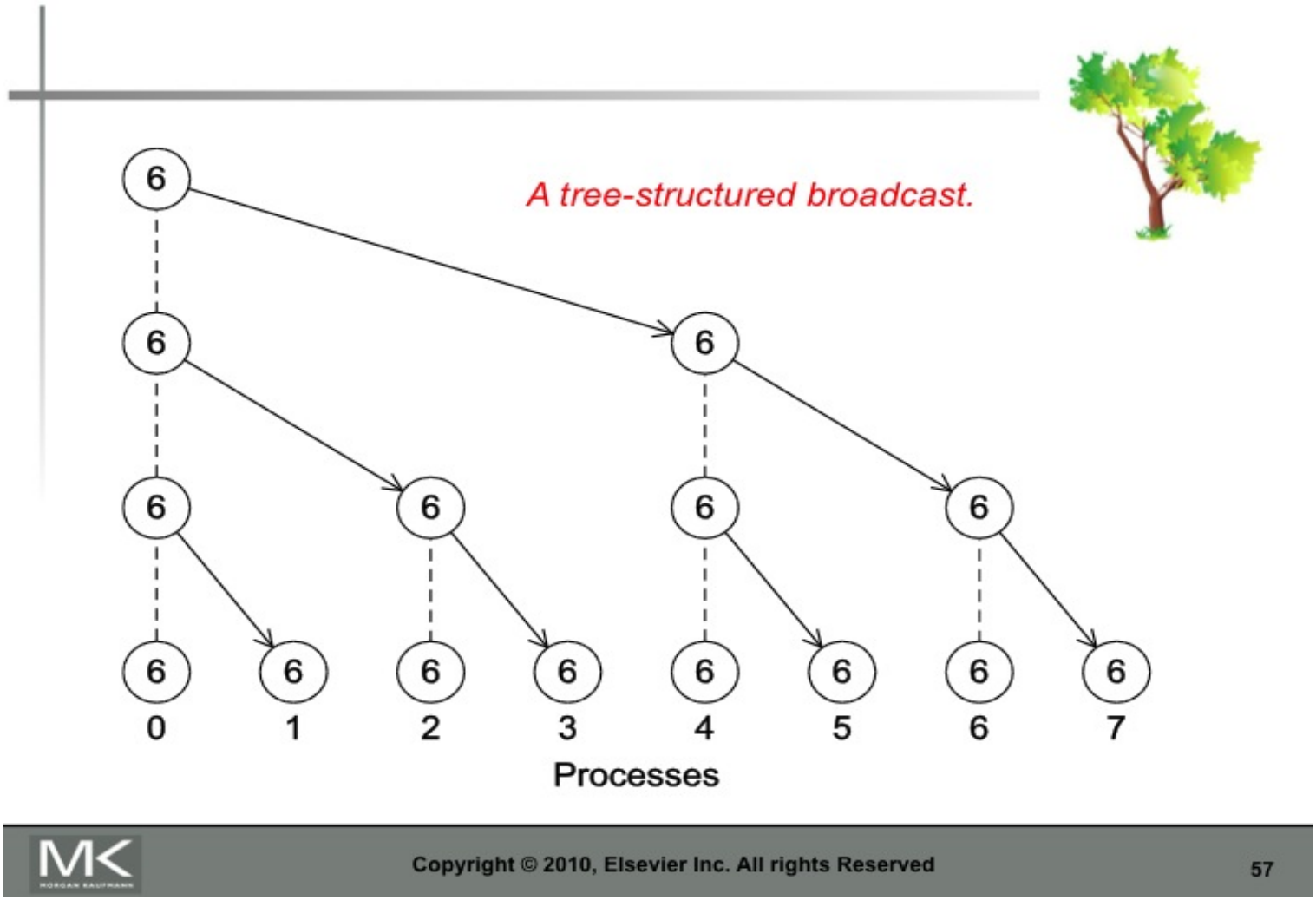
A butterfly-structured global sum.

O log_2 (n)

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

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

Compute a vector sum.

Serial implementation of vector addition

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

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

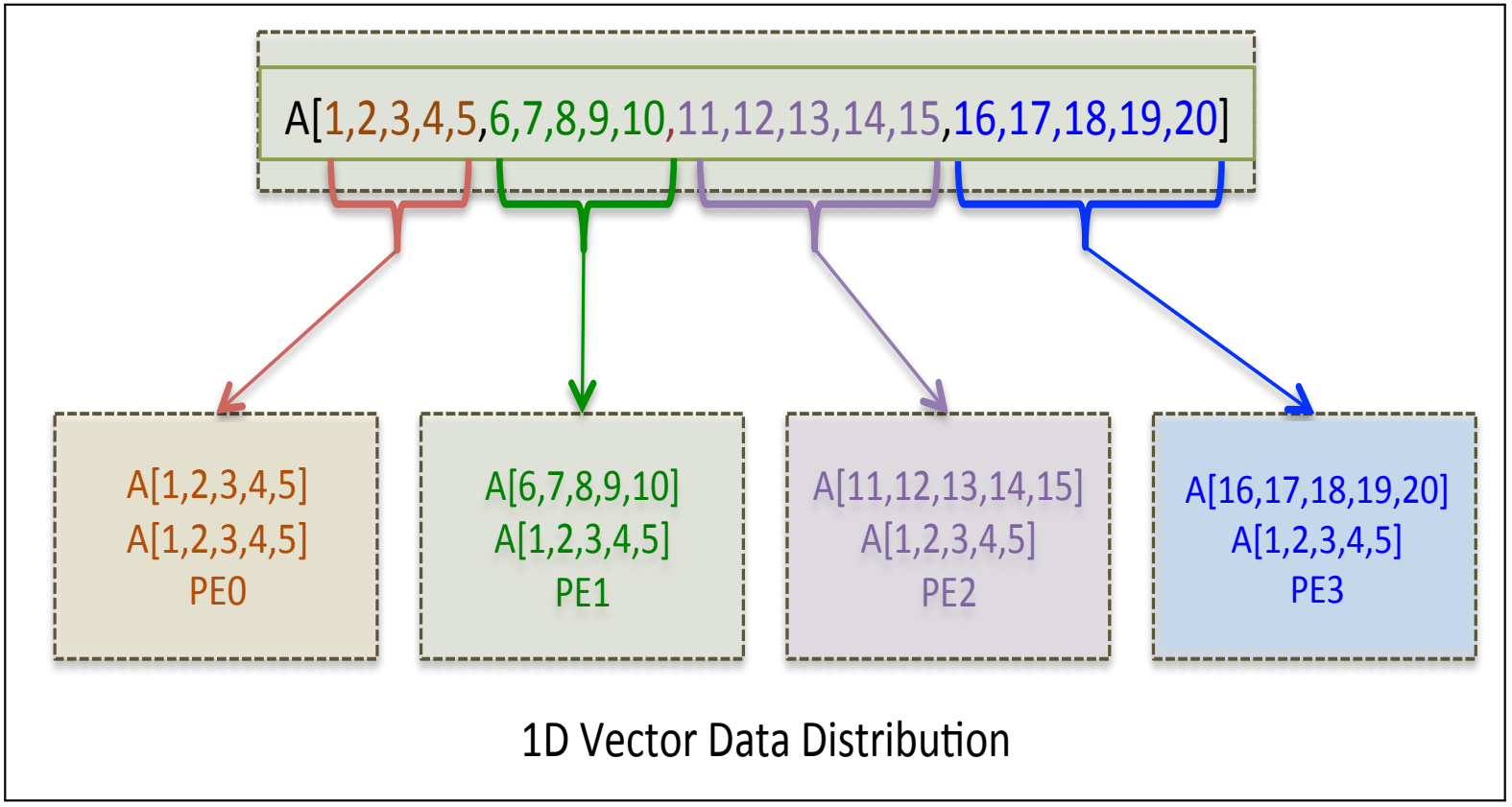
Different partitions of a 12-component vector among 3 processes

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

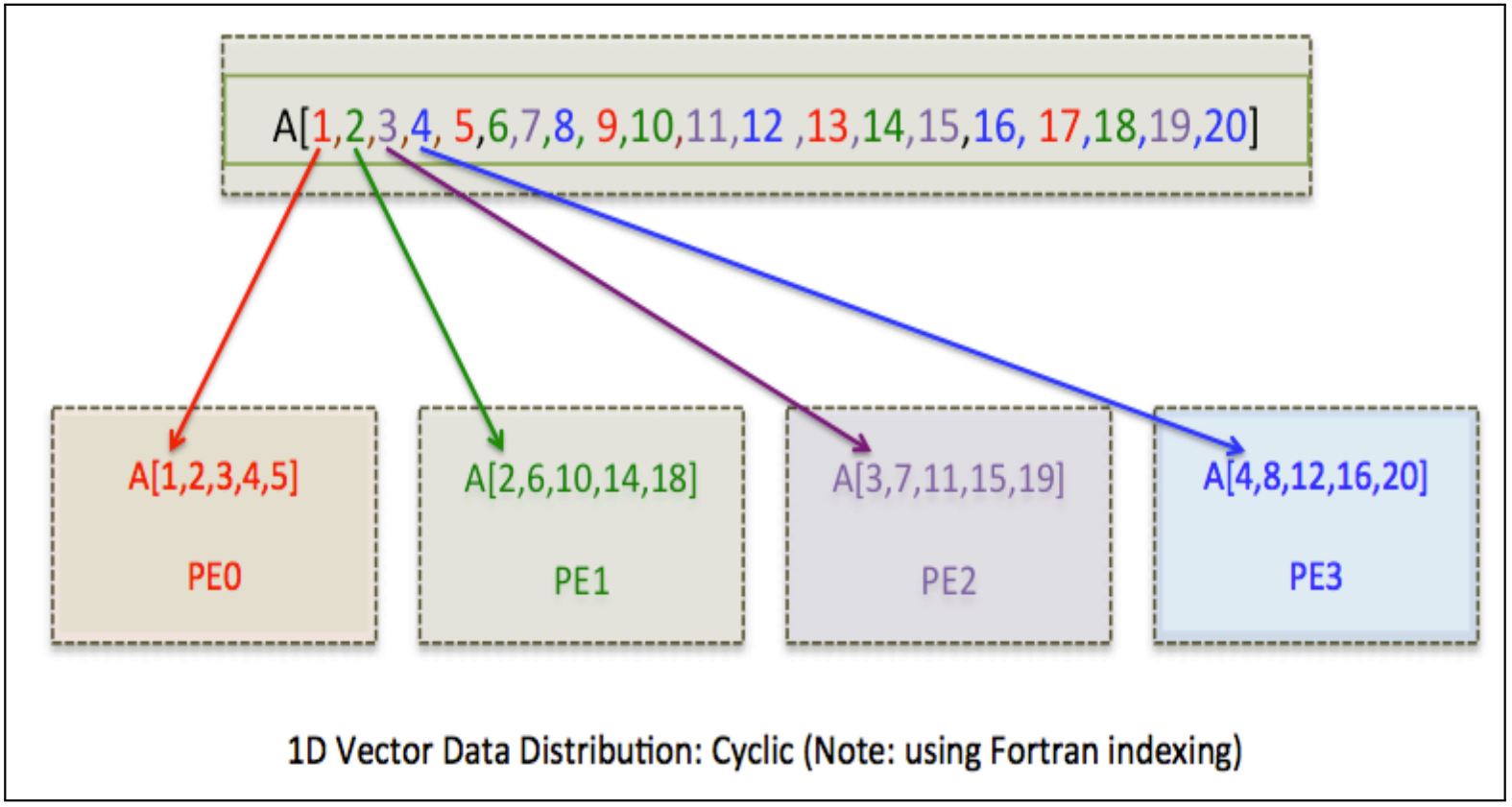
Partitioning options

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

Block Partitioning of a 20 element vector across 4 processors.



Cyclic Partitioning of a 20 element vector across 4 processors.



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  */);
```


Note: the calculation of buffers and counts must be done by programmer.

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 */
```


All nodes call *MPI_Scatter*; only the master node has a value for for *a*; the destination processors store the value of *a* into local variable *local_n*

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  */);
```