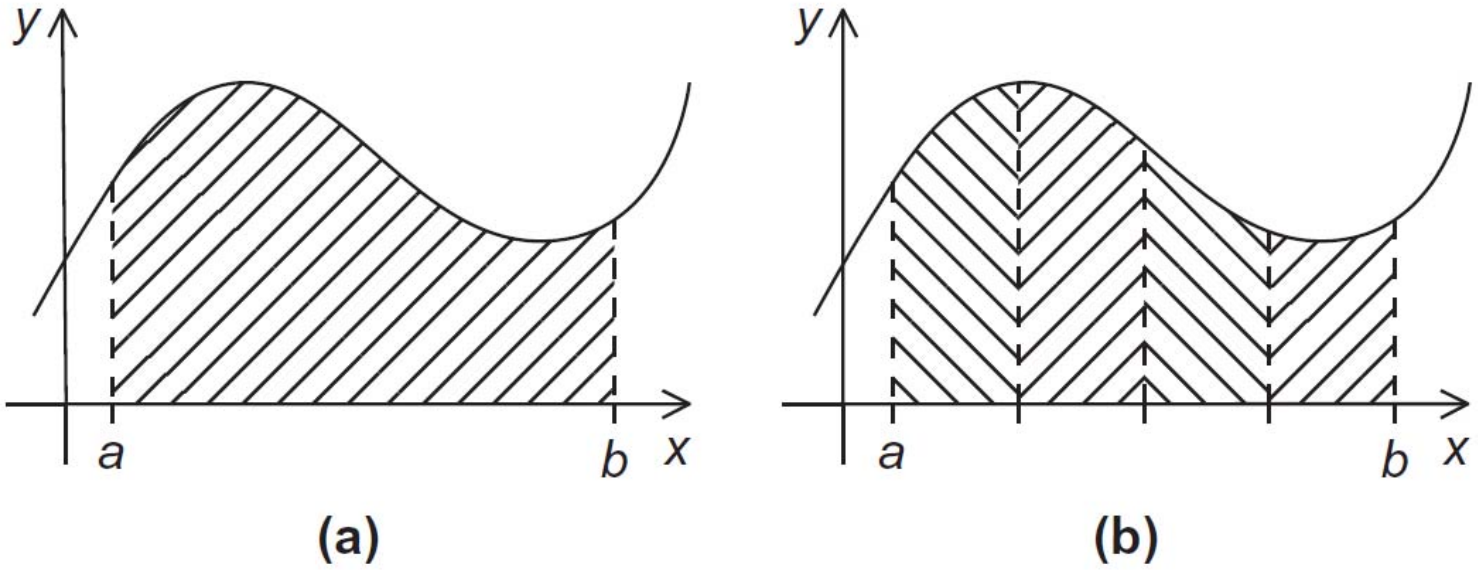
ECE 432/532
Programming for Parallel Processors

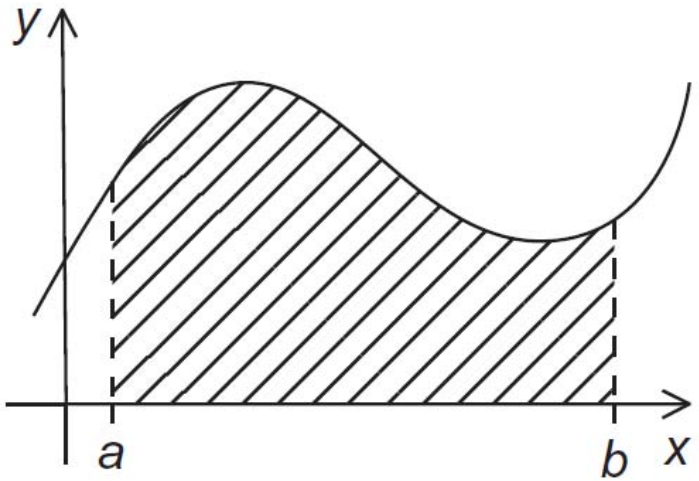
The Trapezoidal Rule

- In mathematics, and more specifically in numerical analysis, the trapezoidal rule is a technique for approximating the definite integral

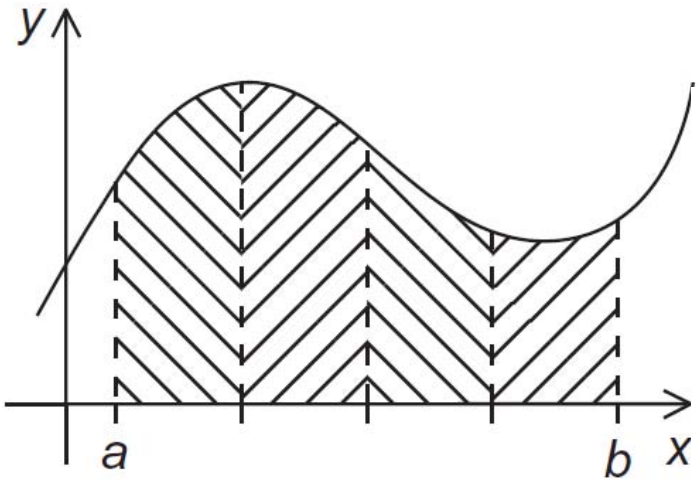


The Trapezoidal Rule

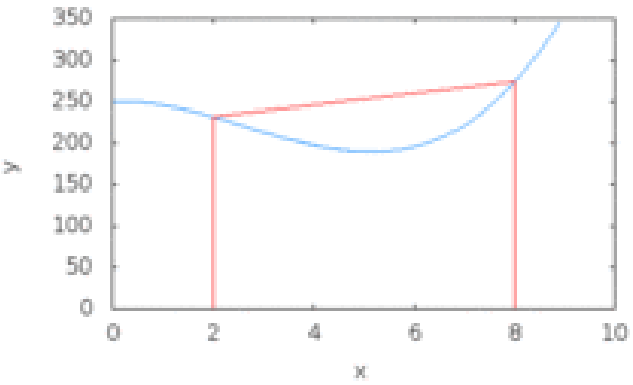
- In mathematics, and more specifically in numerical analysis, the trapezoidal rule is a technique for approximating the definite integral



(a)

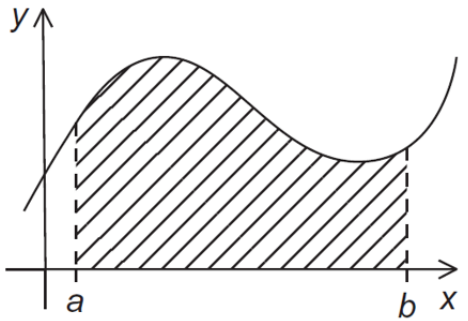


(b)



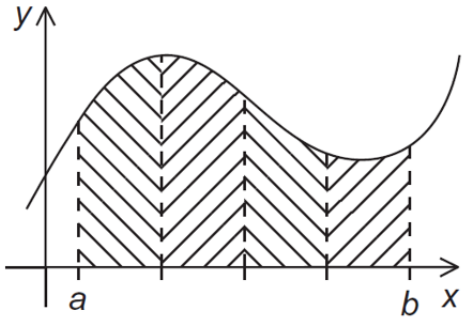
The Trapezoidal Rule

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



Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are x=a and x=b, then

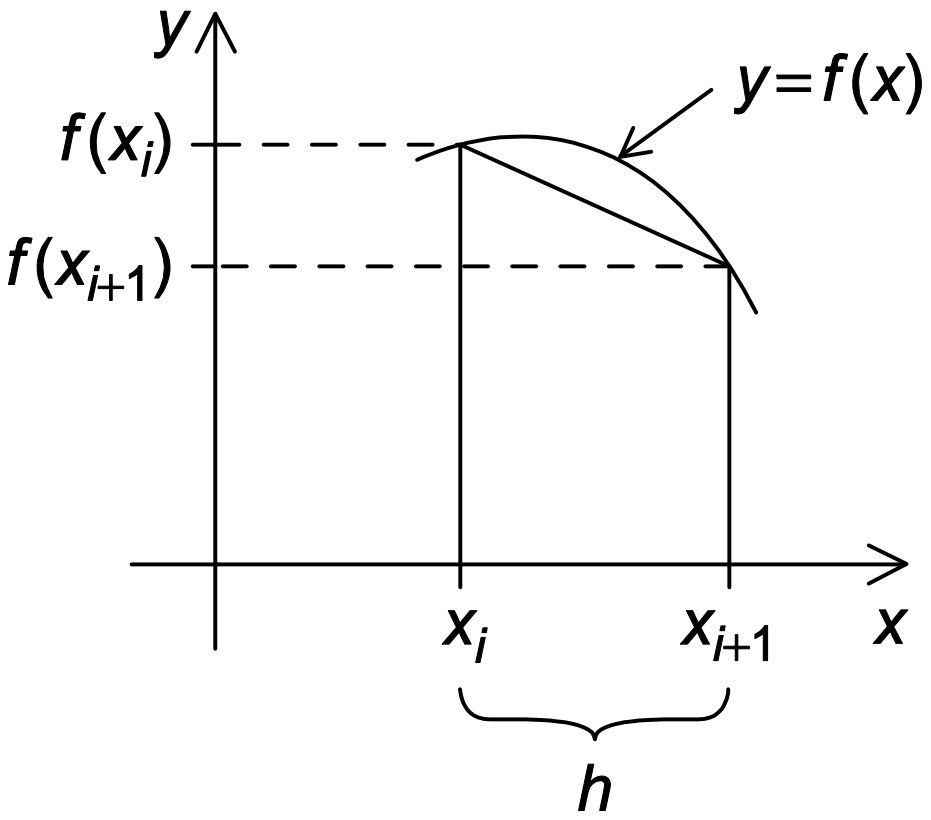
$$h = \frac{b-a}{n}$$



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

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

One trapezoid



Pseudo-code for a serial program

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

Parallelizing the Trapezoidal Rule

- In the partitioning phase, we usually try to identify as many tasks as possible
- For the trapezoidal rule, we might identify two types of tasks
  - Find the area of a single trapezoid
  - Computer the sum of these areas

Parallelizing the Trapezoidal Rule

- So how can we aggregate the tasks and map them to the cores?

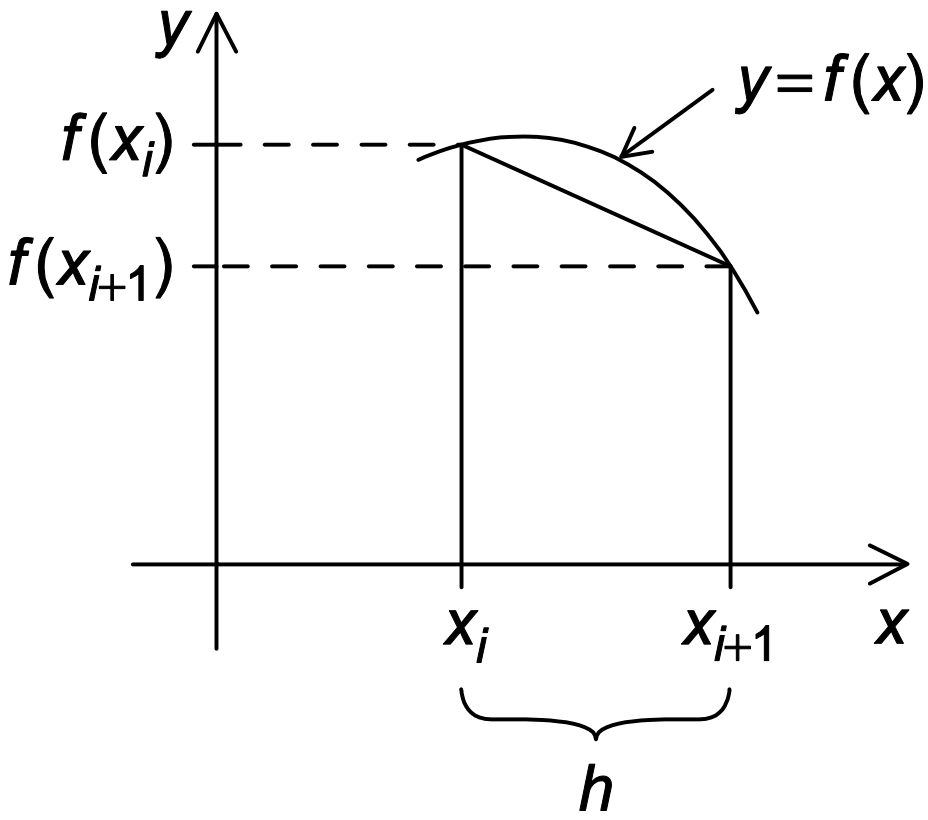
- The more trapezoids we use, the more accurate our estimate will be
  - use many more trapezoids than cores

- We need to aggregate the computation of the areas of the trapezoids into groups:
  - split the interval [*a*,*b*] up into comm_sz subintervals
  - have one of the processes, say process 0, add the estimates

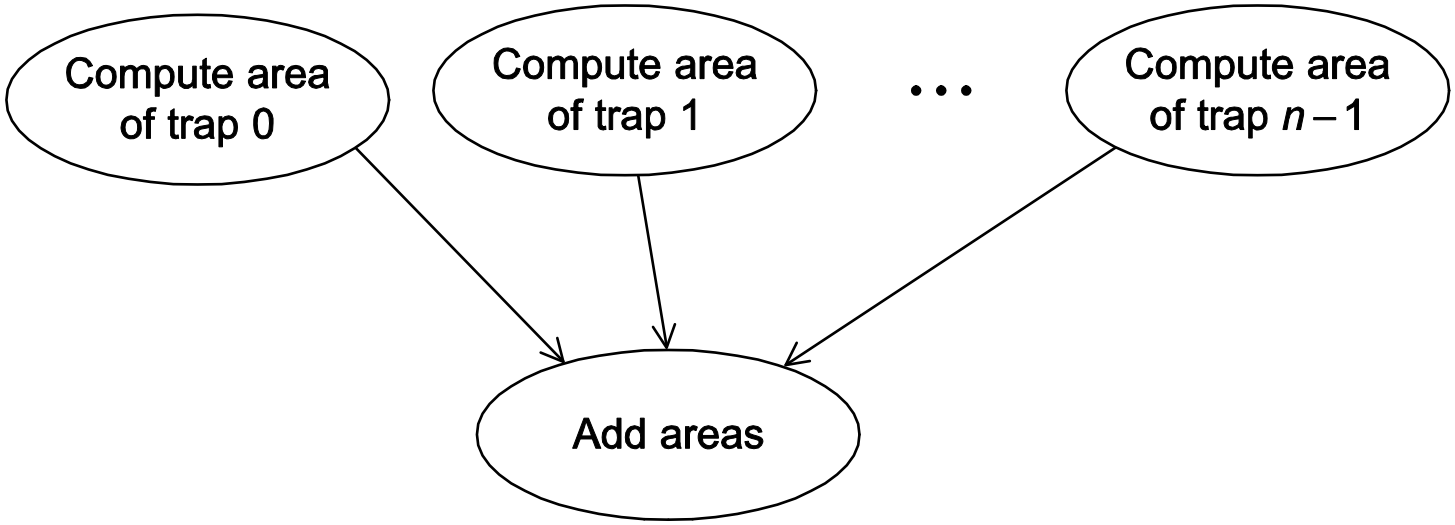
One trapezoid

Pseudo-code for a parallel program

```
Get a, b, n;
h = (b-a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
   Send local_integral to process 0;
else /* my_rank == 0 */
   total_integral = local_integral;
   for (proc = 1; proc < comm_sz; proc++) {
      Receive local_integral from proc;
      total_integral += local_integral;
   }
}
if (my_rank == 0)
   print result;
```



Tasks and communications for Trapezoidal Rule



```
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);
```

```
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  */
```

```
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

- The current version of the parallel trapezoidal rule has a serious deficiency:
  - it will only compute the integral over the interval [0, 3] using 1024 trapezoids

- We can edit the code and recompile → quite a bit of work compared to simply typing in three new numbers

- How can we implement that?

Dealing with I/O

- In both the “greetings” program and the “trapezoidal rule” program we’ve assumed that process 0 can write to stdout

- MPI standard doesn’t specify which processes have access to which I/O devices
  - virtually all MPI implementations allow all the processes in MPI_COMM_WORLD full access to stdout and stderr

- Most MPI implementations don’t provide any automatic scheduling of access to these devices → output are unpredictable

Dealing with I/O

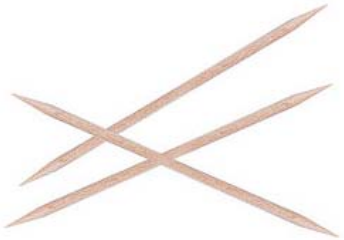
```
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

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

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

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

Each process just prints a message.

Running with 6 processes

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

unpredictable output



Input

- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin
- If multiple processes have access to stdin, which process should get which parts of the input data?
  - E.g. should process 0 get the first line? Process 1 the second?
- Process 0 must read the data (scanf) and send to the other processes.

Input

- In order to write MPI programs that can use scanf, we need to branch on process rank, with process 0 reading in the data and then sending it to the other processes.

- Write a Get_input function for the parallel trapezoidal rule program
  - Process 0 simply reads in the values for *a*, *b*, and *n* and
  - sends all three values to each process.

- This function uses the same basic communication structure as the “greetings” program, except that now process 0 is sending to each process, while the other processes are receiving.

Input

```
. . .
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;
. . .
```

```
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

Collective communication

- Think about the trapezoidal rule program
  - Suggest improvement

- Each process with rank greater than 0 is “telling process 0 what to do” and then it quits
  - Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing

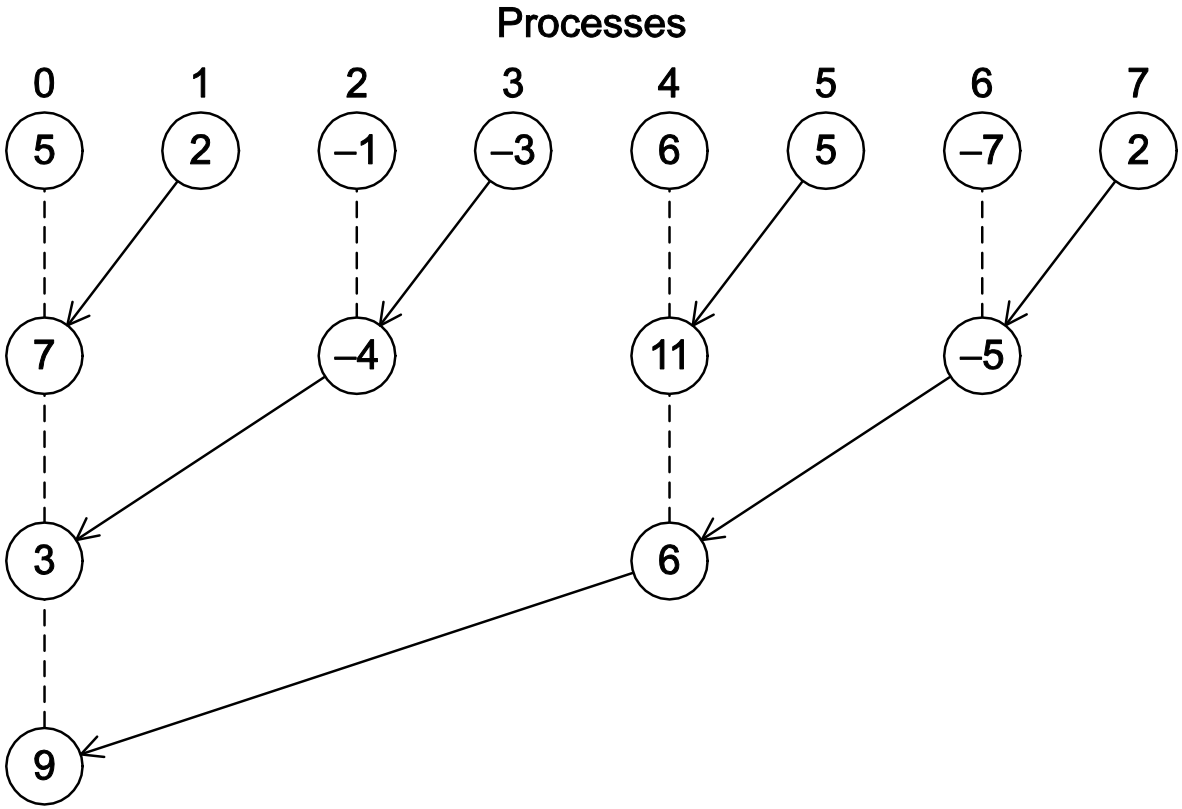
Tree-structured communication

- Use a “binary tree structure”

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

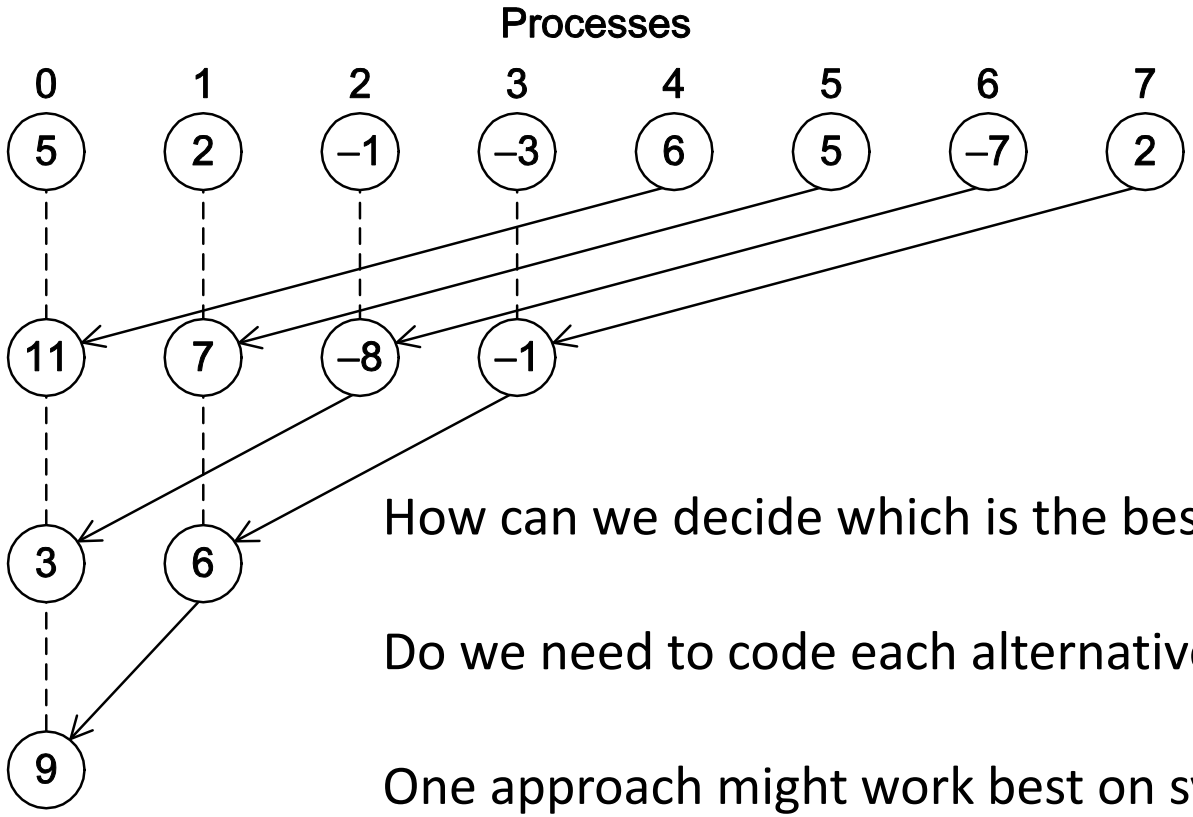


A tree-structured global sum

- if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, that is, three receives and three additions
  - Reduce the overall time by more than 50%.

- If we use more processes, we can do even better
  - For example, if comm_sz =1024, then the original scheme requires process 0 to do 1023 receives and additions, while the new scheme requires process 0 to do only 10 receives and additions

An alternative tree-structured global sum



How can we decide which is the best?

Do we need to code each alternative and evaluate its performance?

One approach might work best on system A, while another might work best on system B.

MPI optimizations

- It’s unreasonable to expect each MPI programmer to write an optimal global-sum function
- MPI specifically protects programmers against this trap of endless optimization by requiring that MPI implementations include implementations of global sums.
- This places the burden of optimization on the developer of the MPI implementation, rather than the application developer
- The assumption here is that the developer of the MPI implementation should know enough about both the hardware and the system software so that she can make better decisions about implementation details

Collective communications

- Now, a “global-sum function” requires communication
- However, unlike the MPI_Send-MPI_Recv pair, the global-sum function may involve more than two processes
  - In the trapezoidal rule program it involves all the processes in MPI_COMM_WORLD
- In MPI, communication functions that involve all the processes in a communicator are called **collective communications**
- MPI_Send and MPI_Recv are often called **point-to-point** communications

MPI_Reduce

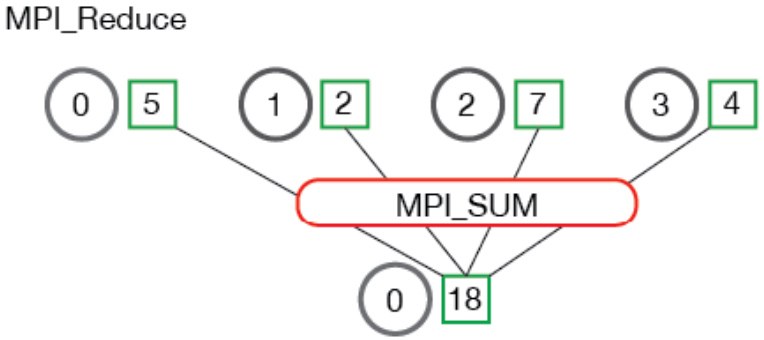
- In fact, global sum is just a special case of an entire class of collective communications

- For example, it might happen that instead of finding the sum of a collection of comm_sz numbers distributed among the processes, we want to find the
  - maximum or
  - the minimum or
  - the product or any one of many other possibilities

- MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function: **MPI_Reduce**

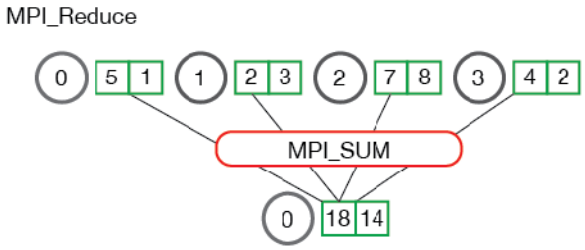
MPI_Reduce

```
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);
```



MPI_Reduce

- The key to the generalization is the fifth argument, operator. It has type MPI_Op
- MPI_Op is a predefined MPI type like MPI_Datatype and MPI_Comm
- There are a number of predefined values in this type
  - It’s also possible to define your own operators

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.

- 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

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

- What will be the values of b and d?

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, ...)`

Example

- 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

- A final caveat: it might be tempting to call MPI Reduce using the same buffer for both input and output. For example

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

- However, this call is illegal in MPI

```
int main(int argc, char** argv)
{
    int size, rank,i=0,localsum1=0,globalsum1=0,localsum2=0,globalsum2=0;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank==0){
    }
    else if(rank==1) {
        localsum1 += 5;
        MPI_Reduce(&localsum1,&globalsum1,2,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
    }
    else if(rank==2){
        localsum2 += 10;
        MPI_Reduce(&localsum2,&globalsum2,2,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
    }

    if(rank==0) {
        printf("globalsum1 = %d \n",globalsum1);
        printf("globalsum2 = %d \n",globalsum2);
    }
    MPI_Finalize();
    return (EXIT_SUCCESS);
}
```

Find the error

Solution

- MPI_Reduce is a collective operation → all tasks in the participating communicator must make the MPI_Reduce() call.
  - In the above, rank 0 never calls MPI_Reduce() so this program will hang as some of the other processors wait for participation from rank 0 which will never come.

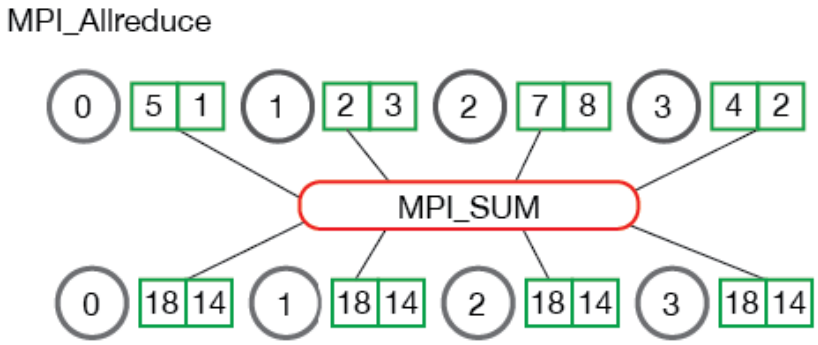
- Also, because it is a collective operation on the entire communicator, you need to do some work to partition the reduction.
  - One way is just to reduce an array of ints, and have each processor contribute only to its element in the array:

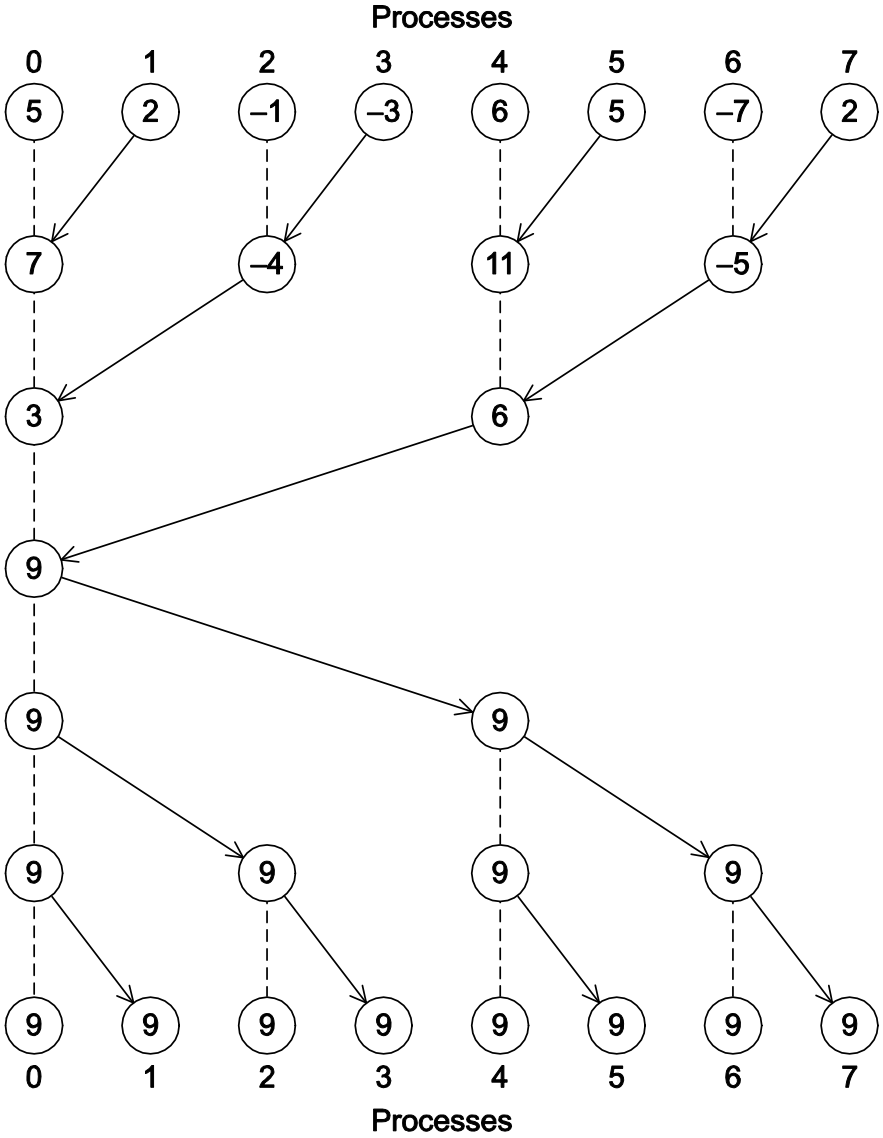
```
int main(int argc, char** argv)
{
    int size, rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    int localsum[2] = {0,0};
    int globalsum[2] = {0,0};

    if(rank % 2 == 1){
        localsum[0] += 5;
    }
    else if( rank > 0 && (rank % 2 == 0)){
        localsum[1] += 10;
    }

    MPI_Reduce(localsum,globalsum,2,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

    if(rank==0)   {
        printf("globalsum1 = %d \n",globalsum[0]);
        printf("globalsum2 = %d \n",globalsum[1]);
    }

    MPI_Finalize();
    return (EXIT_SUCCESS);
}
```

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.

- if we use a tree to compute a global sum, we might “reverse” the branches to distribute the global sum

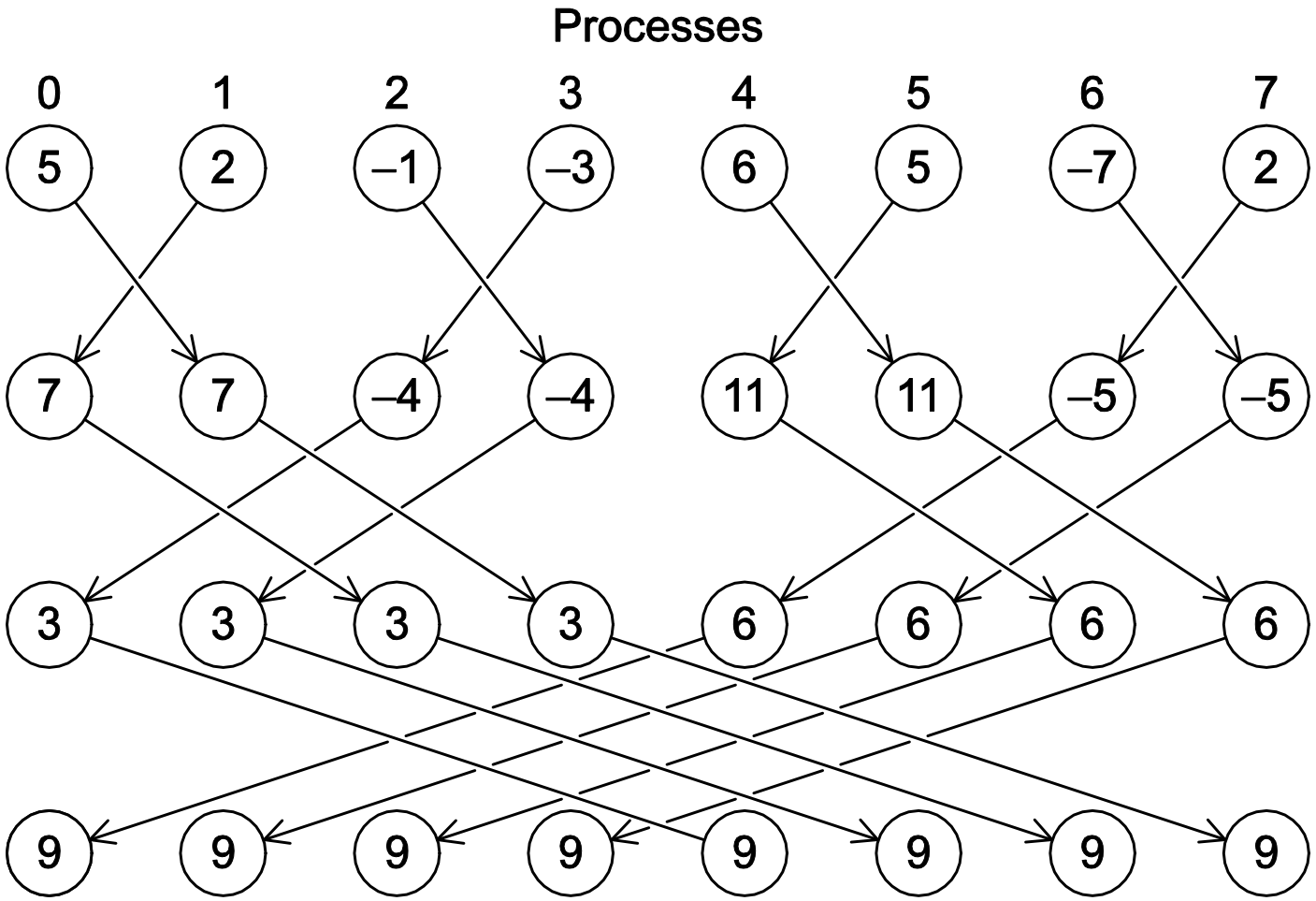




A global sum followed by distribution of the result.

MPI_Allreduce

- Alternatively, we might have the processes *exchange* partial results instead of using one-way communications
- Such a communication pattern is sometimes called a **butterfly**
- MPI provides a variant of MPI_Reduce that will store the result on all the processes in the communicator: MPI_Allreduce



A butterfly-structured global sum.

MPI_Allreduce

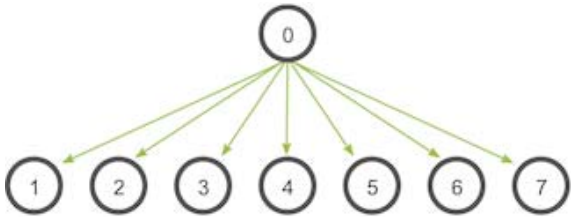
- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```
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  */);
```

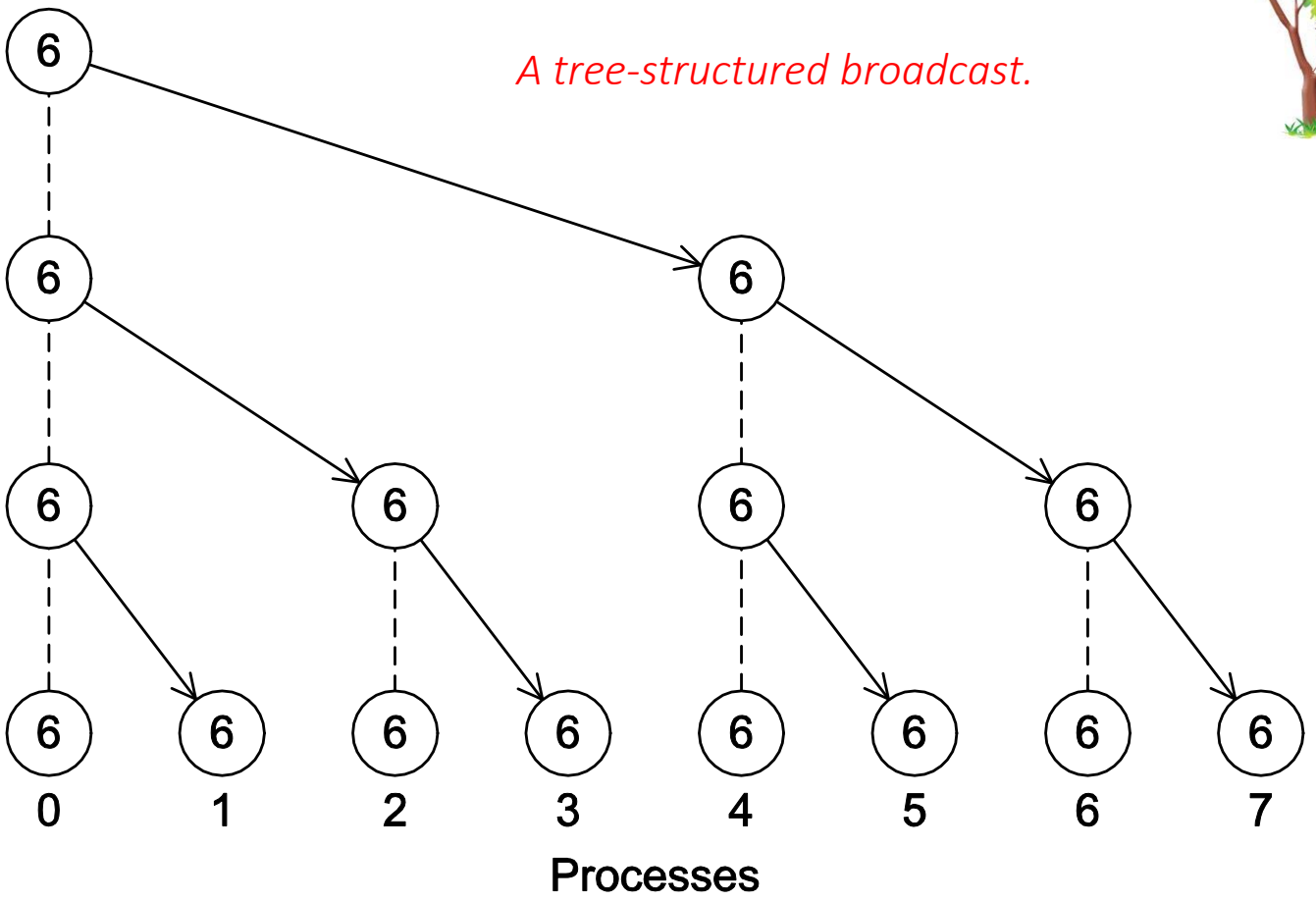
Broadcast

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

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



A tree-structured broadcast.



A version of Get_input that uses MPI_Bcast

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

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

```
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 */
```

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

int main(int argc, char** argv) {
        int rank;
        int buf;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if(rank == 0) {
                buf = 777;
                MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
        }
        else {
                MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
                printf("rank %d receiving received %d\n", rank, buf);
        }

        MPI_Finalize();
        return 0;
}
```

Find the error

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

int main(int argc, char** argv) {
        int rank;
        int buf;
        const int root=0;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if(rank == root) {
           buf = 777;
        }

        printf("[%d]: Before Bcast, buf is %d\n", rank, buf);

        /* everyone calls bcast, data is taken from root and ends up in everyone's buf */
        MPI_Bcast(&buf, 1, MPI_INT, root, MPI_COMM_WORLD);

        printf("[%d]: After Bcast, buf is %d\n", rank, buf);

        MPI_Finalize();
        return 0;
}
```

```
#include <stdio.h>
#include <mpi.h>     /* For MPI functions, etc */

int main(void) {
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int a[10];
   int b[10]={1,2,3,4,5,6,7,8,9,10};
   int c[10]={1,2,3,4,5,6,7,8,9,10};

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

   MPI_Reduce(b,c,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0) {
        for (int q = 0; q < 10; q++) {
                printf("%d\n", c[q]);
      }
   }

MPI_Finalize();

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

What will it print with mpiexec –n 3 ./test

```
#include <stdio.h>
#include <mpi.h>     /* For MPI functions, etc */

int main(void) {
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int a[10];
   int b[10]={1,2,3,4,5,6,7,8,9,10};
   int c[10]={1,2,3,4,5,6,7,8,9,10};

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

   MPI_Reduce(b,c,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0) {
        for (int q = 0; q < 10; q++) {
                printf("%d\n", c[q]);
      }
   }

MPI_Finalize();

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

What will it print with mpiexec –n 10 ./test

```
#include <stdio.h>
#include <mpi.h>     /* For MPI functions, etc */

int main(void) {
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int a[10];
   int b[10]={1,2,3,4,5,6,7,8,9,10};
   int c[10]={1,2,3,4,5,6,7,8,9,10};

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

   MPI_Reduce(b,c,3,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0) {
        for (int q = 0; q < 10; q++) {
                printf("%d\n", c[q]);
      }
   }

MPI_Finalize();

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

What will it print with mpiexec –n 2 ./test

```
#include <stdio.h>
#include <mpi.h>     /* For MPI functions, etc */

int main(void) {
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int a[10];
   int b[10]={1,2,3,4,5,6,7,8,9,10};
   int c[10]={1,2,3,4,5,6,7,8,9,10};

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

   MPI_Reduce(b,c,3,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0) {
        for (int q = 0; q < 10; q++) {
                printf("%d\n", c[q]);
      }
   }

MPI_Finalize();

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

What will it print with mpiexec –n 10 ./test

```
#include <stdio.h>
#include <mpi.h>     /* For MPI functions, etc */

int main(void) {
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int a[10];
   int b[10]={1,2,3,4,5,6,7,8,9,10};
   int c[10]={1,2,3,4,5,6,7,8,9,10};

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

   MPI_Reduce(b,c,10,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0) {
        for (int q = 0; q < 10; q++) {
                printf("%d\n", c[q]);
      }
   }

MPI_Finalize();

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

What will it print with mpiexec –n 1 ./test

```
#include <stdio.h>
#include <mpi.h>     /* For MPI functions, etc */

int main(void) {
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */
   int a[10];
   int b[10]={1,2,3,4,5,6,7,8,9,10};
   int c[10]={1,2,3,4,5,6,7,8,9,10};

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

   MPI_Reduce(b,c,10,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0) {
        for (int q = 0; q < 10; q++) {
                printf("%d\n", c[q]);
      }
   }

MPI_Finalize();

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

What will it print with mpiexec –n 10 ./test

Data distributions

- Suppose we want to write a function that computes a vector sum:

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

Data distributions

- If we implement the vectors as arrays of **double**s
- The serial vector addition is:

```
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 */
```

Data distributions

- How could we implement this using MPI?
  - The work consists of adding the individual components of the vectors

- Then there is no communication between the tasks → aggregate the tasks and assigning them to the cores

- If the number of components is *n* and we have comm_sz cores or processes
  - assign blocks of local_n = n/comm_sz consecutive components to each process

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.

Data distributions

- *n*=12 and comm_sz = 3

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

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

- It would be convenient to be able to read the dimension of the vectors and then read in the vectors **x** and **y**
  - process 0 can prompt the user, read in the value, and broadcast the value to the other processes

- However, this could be very wasteful
  - If there are 10 processes and the vectors have 10,000 components
  - each process will need to allocate storage for vectors with 10,000 components
  - it is only operating on subvectors with 1000 components.

Scatter

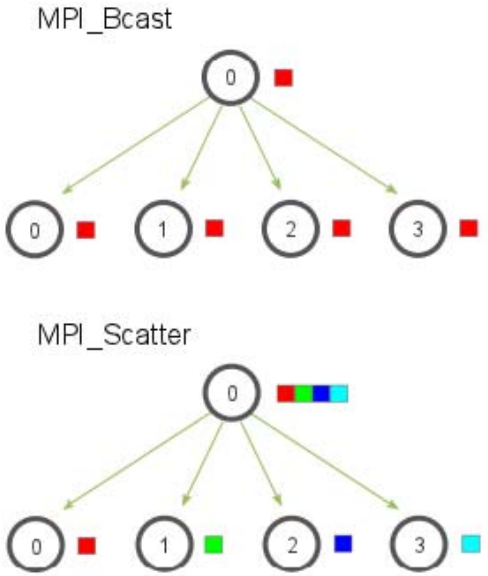
- In a block distribution, it would be better
  - process 0 sends only components 1000 to 1999 to process 1,
  - components 2000 to 2999 to process 2, and so on

- Using this approach, processes 1 to 9 would only need to allocate storage for the components they’re actually using

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

- divides the data referenced by send_buf_p into comm_sz pieces
- Each process should pass its local vector as recv_buf_p
- recv_count argument should be local_n
- send_count is the *amount of data going to each process* (usualy local_n)

```
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

- The first parameter, is an array of data that resides on the root process
- The second and third parameters, dictate how many elements of a specific MPI Datatype will be sent to each process
- If send_count is two, then process zero gets the first and second integers, process one gets the third and fourth, and so on. In practice, send_count is often equal to the number of elements in the array divided by the number of 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 */
```

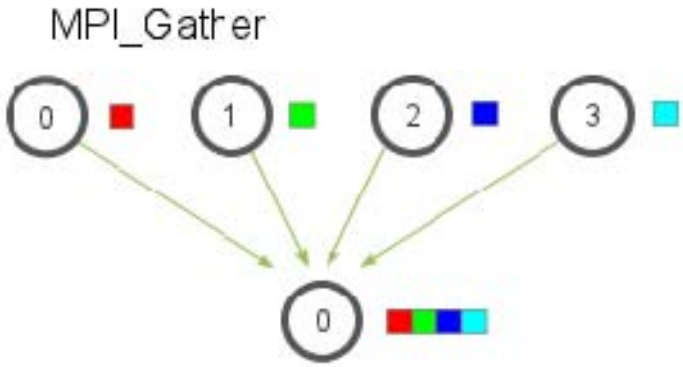
MPI_Gather

- Our program will be useless unless we can see the result of our vector addition
- We need a function for printing out a distributed vector
- The function should collect all of the components of the vector onto process 0, and then process 0 can print all of the components
- The communication in this function can be carried out by MPI_Gather

MPI_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

- The data stored in the memory referred to by send_buf_p on process 0 is stored in the first block in recv_buf-p
- The data stored in the memory referred to by send_buf_p on process 1 is stored in the second block referred to by recv_buf_p, and so on

```
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  */);
```

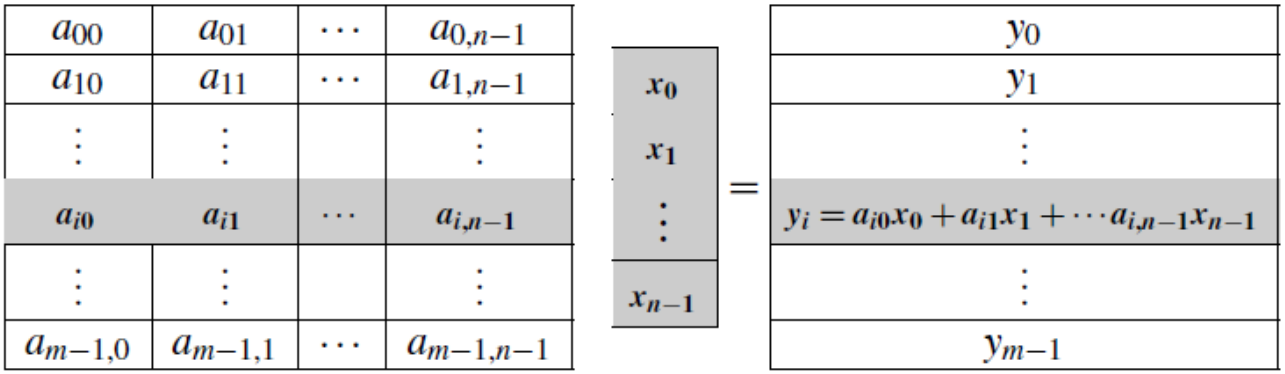
- recv_count is the number of data items received from *each* process, not the total number of data items received

```
 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 */
```

Gathering results from everybody

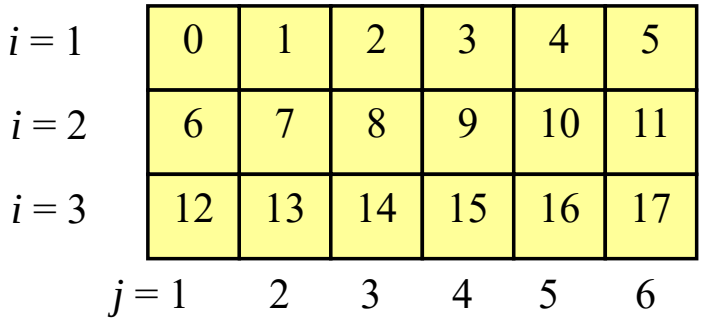
- Write an MPI function that multiplies a matrix by a vector **y=*A*x**

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



Gathering results from everybody

- C programmers frequently use one-dimensional arrays to “simulate” two-dimensional arrays



$n = 3$

$m = 6$

$loc\,(n, m, i, j) = m(i-1) + j-1$

```
y[i] += A[i*n+j]*x[j];
```

Matrix-vector multiplication

```
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 */
```

- So if x has a block distribution, how can we arrange that each process has access to *all* the components of x before we execute the following loop

MPI_Allgather

- Using the collective communications, we could execute a call to MPI_Gather followed by a call to MPI_Bcast

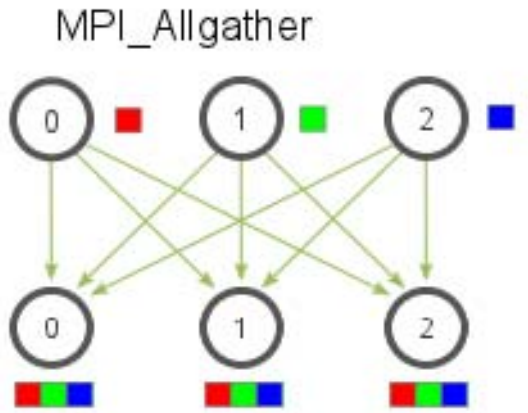
- This would, in all likelihood, involve two tree-structured communications, and we may be able to do better by using a butterfly

- MPI provides a single function: MPI_Allgather

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.

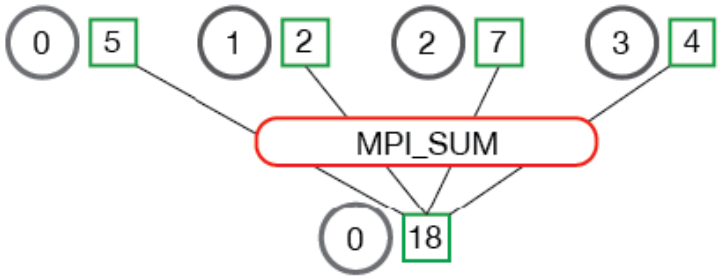
```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm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



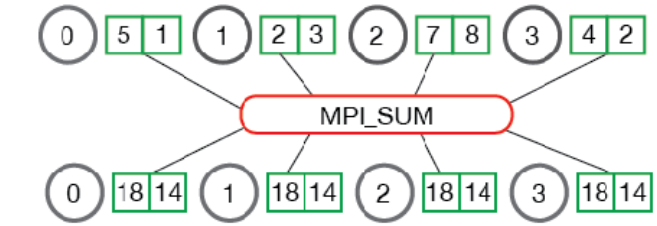
```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* 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 */
```

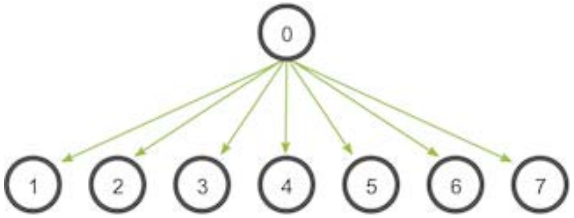
So far …

MPI_Reduce

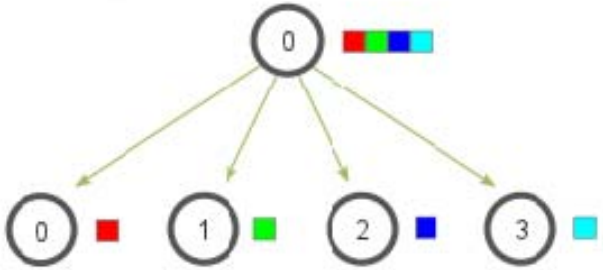


MPI_Allreduce

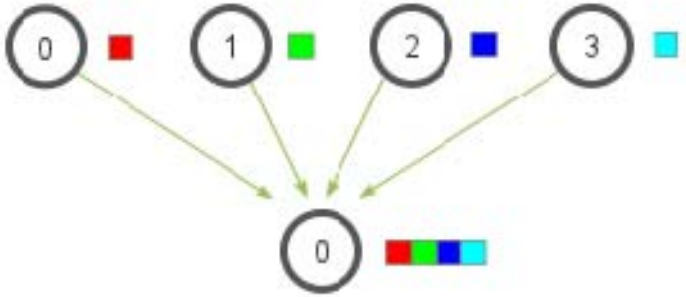




MPI_Scatter



MPI_Gather



MPI_Allgather