Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

- MPI – SPMD Model
- How to to compile an MPI program (mpicc) and run N processes in parallel (mpirun)
- How to retrieve the number of running processes (MPI_Comm_size)
- How to retrieve my ID (MPI_Comm_rank)
- How to send (MPI_Send) and receive (MPI_Recv) data
- How to structure a parallel application

- On the 3 hours lectures, we'll have lesson + hands-on sessions (bring the laptop)

Recap

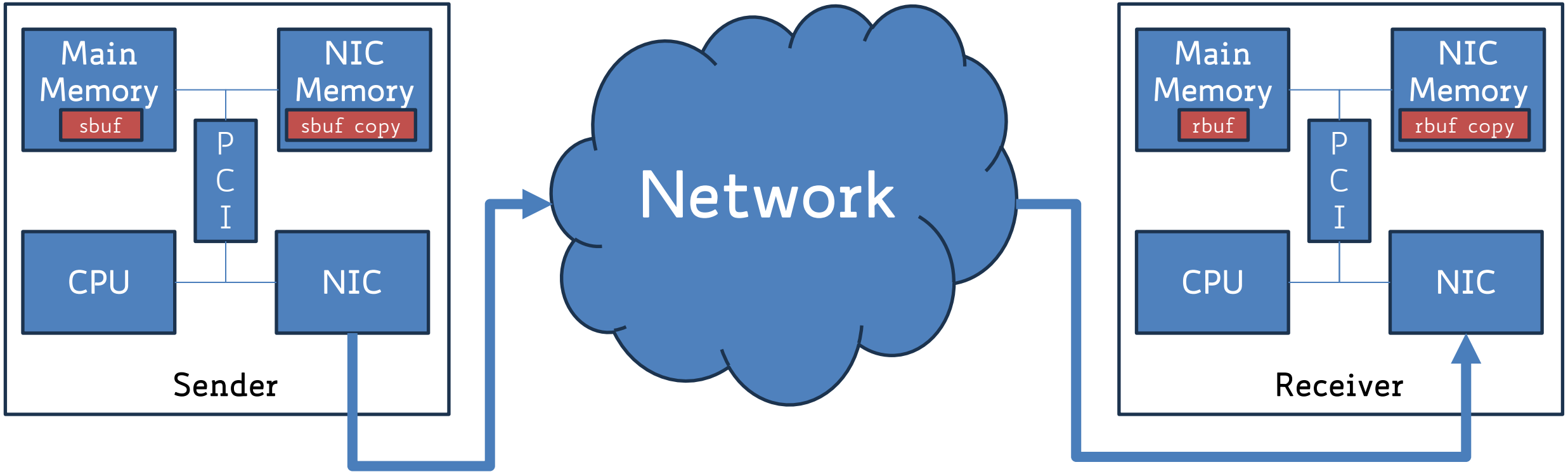
- MPI – SPMD Model
- How to to compile an MPI program (mpicc) and run N processes in parallel (mpirun)
- How to retrieve the number of running processes (MPI_Comm_size)
- How to retrieve my ID (MPI_Comm_rank)
- How to send (MPI_Send) and receive (MPI_Recv) data
- How to structure a parallel application

- On the 3 hours lectures, we'll have lesson + hands-on sessions (bring the laptop)

What happens when you do a Send?

MPI_Send(sbuf, …)

MPI_Recv(rbuf, …)



What happens when you do a Send?

MPI_Send(buf, 100, …)

MPI_Send(buf + 100, 100,…)

vs.

MPI_Send(buf, 200,…)

1 Send of 200 bytes better than

2 Sends of 100 bytes each

Chapter 5.7-5.9

Point-to-Point Communication Modes

- MPI_Send uses the so called **standard** communication mode. MPI decides based on the size of the message, whether to block the call until the destination process collects it or to return before a matching receive is issued. The latter is chosen if the message is small enough, making MPI_Send **locally blocking.**
- There are three additional communication modes:
  - **Buffered**: in buffered mode the sending operation is always locally blocking, i.e. it will return as soon as the message is copied to a buffer. The second difference with the standard communication mode is that the buffer is user-provided.
  - **Synchronous**: in synchronous mode, the sending operation will return only after the destination process has initiated and started the retrieval of the message. This is a proper **globally blocking** operation. **Why?**
    - **Sender can be sure of the point where the receiver is without any further explicit communication**
  - **Ready**: the send operation will succeed only if a matching receive operation has been initiated already. Otherwise the function returns with an error code. The purpose of this mode is to reduce the overhead of handshaking operations.

Point-to-Point Communication Modes

```
int [ MPI_Bsend | MPI_Ssend | MPI_Rsend ] (void *buf, int count, ↩
    MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```

Questions?

Non-Blocking Communication

- Buffered sends are considered bad for performance, because the caller has to block, waiting for the copy to take place.
- Non-blocking or **immediate** functions, **maximize concurrency** by returning immediately upon initiating a transfer, allowing **communication and computation to overlap.**
  - **i.e., while the copying is being done and/or the NIC is sending/receiving the data, I can compute something else**
- There are both send and receive immediate variants.

Non-Blocking Communication

- The downside is that the completion of the operations for both end-points, has to be queried explicitly:
  - For senders so that they can re-use or modify the message buffer.
  - For receivers so that they can extract the message contents.
- Non-blocking communications can be coupled with any communication mode: MPI_Isend, MPI_Ibsend, MPI_Issend, etc.

Non-Blocking Send

```
int MPI_Isend(void          *buf,     // Address of data buffer (OUT)
              int           count,    // Number of data items (IN)
              MPI_Datatype  datatype,// Same as in MPI_Send (IN)
              int           source,   // Rank of destination proc. (IN)
              int           tag,      // Label identifying the type
                                      //   of message (IN)
              MPI_Comm      comm,     // Identifies the communicator
                                      //   context of 'source' (IN)
              MPI_Request   *req      // Used to return a handle for
                                      //   checking status (OUT)
             )
```

- The MPI_Request that is returned, is a handle that allows a query on the status of the operation to take place

Non-Blocking Recv

```
int MPI_Irecv(void         *buf,     // Address of receive buff. (OUT)
              int          count,    // Buffer capacity in items (IN)
              MPI_Datatype datatype, // Same as in MPI_Send (IN)
              int          source,   // Rank of sending process (IN)
              int          tag,      // Label identifying the type
                                     //   of message expected (IN)
              MPI_Comm     comm,     // Identifies the communicator
                                     //   context of 'source' (IN)
              MPI_Request  *req      // Used to return a handle for
                                     //   checking status (OUT)
             )
```

- The MPI_Request that is returned, is a handle that allows a query on the status of the operation to take place.
- In MPI_Irecv the MPI_Status parameter is replaced by a MPI_Request one.

Check for Completion

- Blocking (destroys handle):

```
int MPI_Wait(MPI_Request *req, // Address of the handle identifying
                               //   the operation queried (IN/OUT)
                               //   The call invalidates *req by
                               //   setting it to MPI_REQUEST_NULL.
             MPI_Status  *st   // Address of the structure that will
                               //   hold the comm. information (OUT)
            )
```

- Non-blocking (destroys handle if operation was successfull i.e. *flag=1):

```
int MPI_Test(MPI_Request *req, // Address of the handle identifying
                               //   the operation queried (IN)
             int         *flag,// Set to true if operation is
                               //   complete (OUT).
             MPI_Status  *st   // Address of the structure that will
                               //   hold the comm. information (OUT)
            )
```

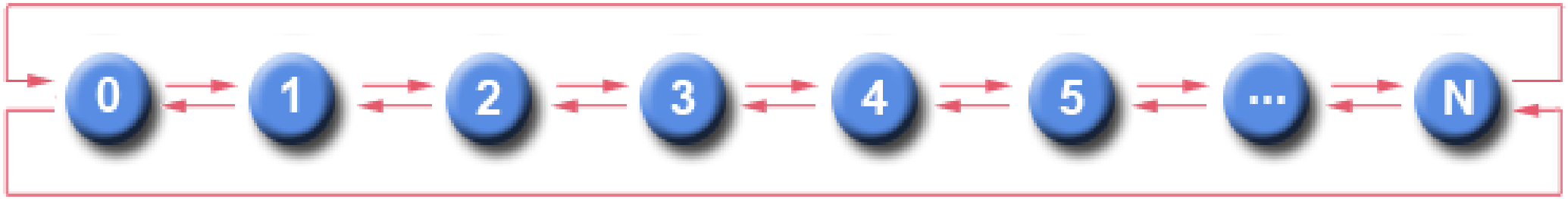
Check for Completion (2)

Several variants available (check man):

- Waitall
- Waitany
- Testany
- etc…

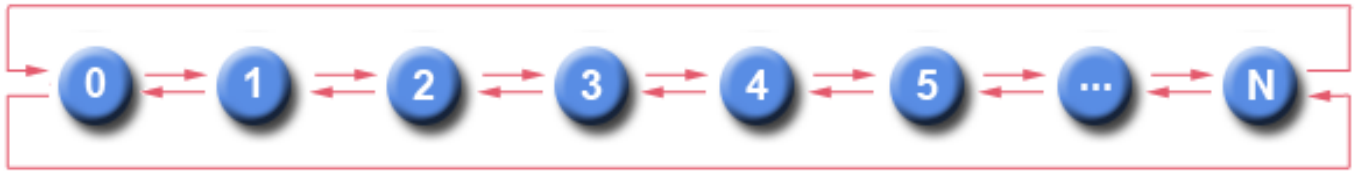
Non-Blocking Comm. Example

- *Problem*: **Ring:** Each rank sends something to left/right rank, and receives something from them



Non-Blocking Comm. Example

- *Problem*: **Ring:** Each rank sends something to left/right rank, and receives something from them




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

int main(void) {
    int numtasks, rank, next, prev, buf[2];
    MPI_Request reqs[4]; // required variable for non-blocking calls
    MPI_Status stats[4]; // required variable for Waitall routine
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // determine left and right neighbors
    prev = (rank-1) % numtasks;
    next = (rank+1) % numtasks;
    // post non-blocking receives and sends for neighbors
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend(&rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[3]);
    // do some work while sends/receives progress in background

    // wait for all non-blocking operations to complete
    MPI_Waitall(4, reqs, stats);
    // continue - do more work
    MPI_Finalize();
}
```

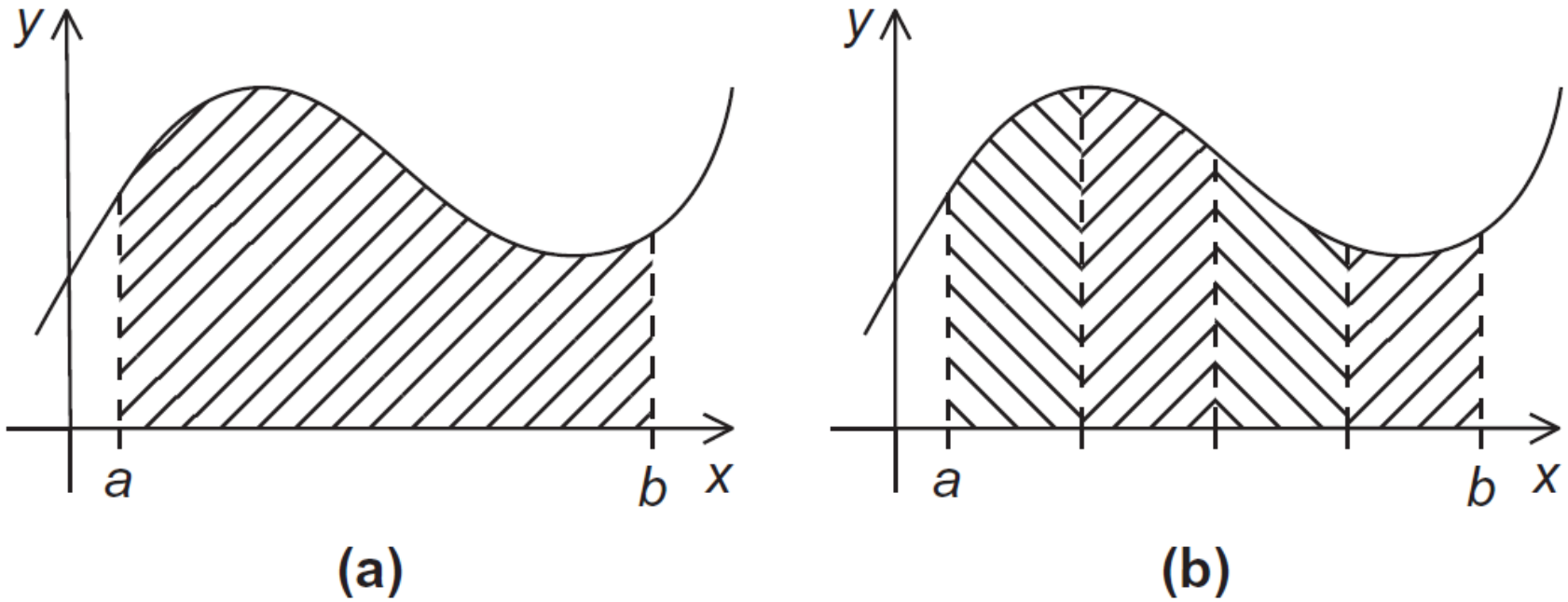
Point-to-point Communication Summary

A sending process ...	**Function**
... must block until the message is delivered	`MPI_Ssend`
... should wait only until the message is buffered	`MPI_Bsend`
... should return immediately without ever blocking	`MPI_Isend`

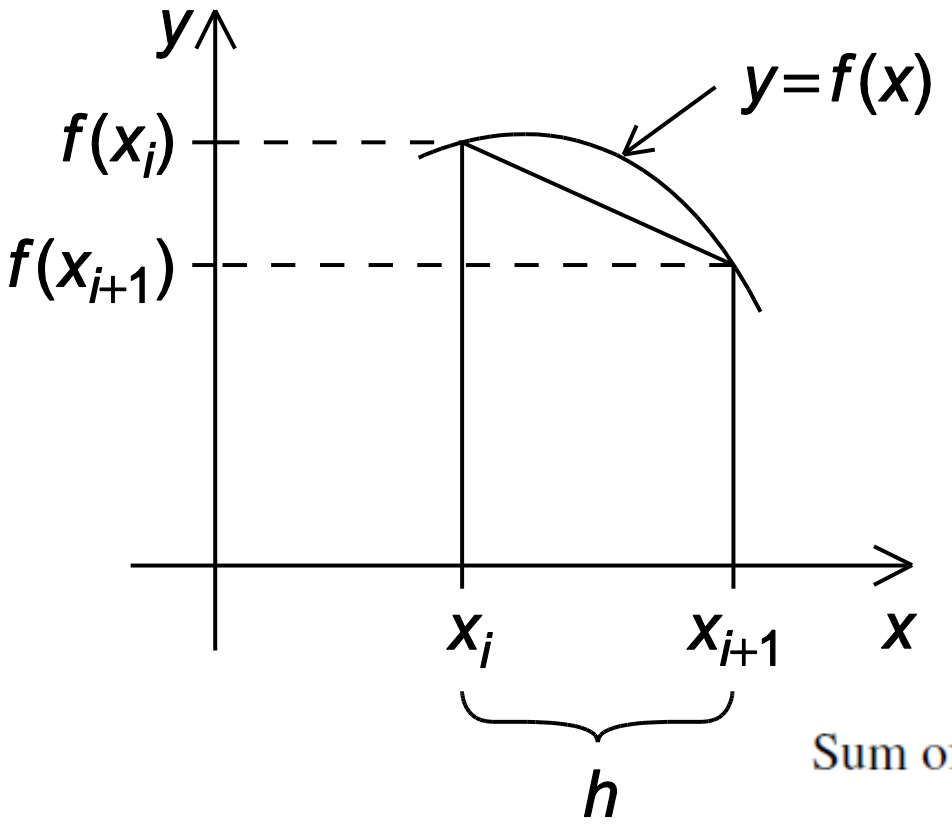
Questions?

Example: Trapezoidal rule in MPI

The Trapezoidal Rule



The Trapezoidal Rule



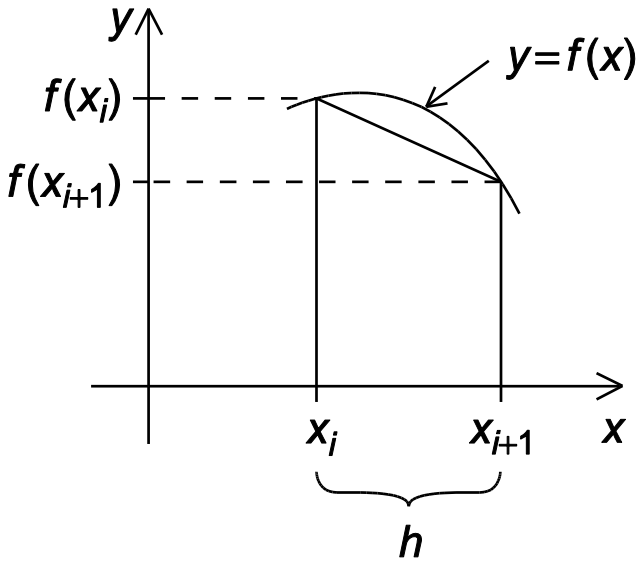
$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

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

Pseudo-code for a serial program



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

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

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

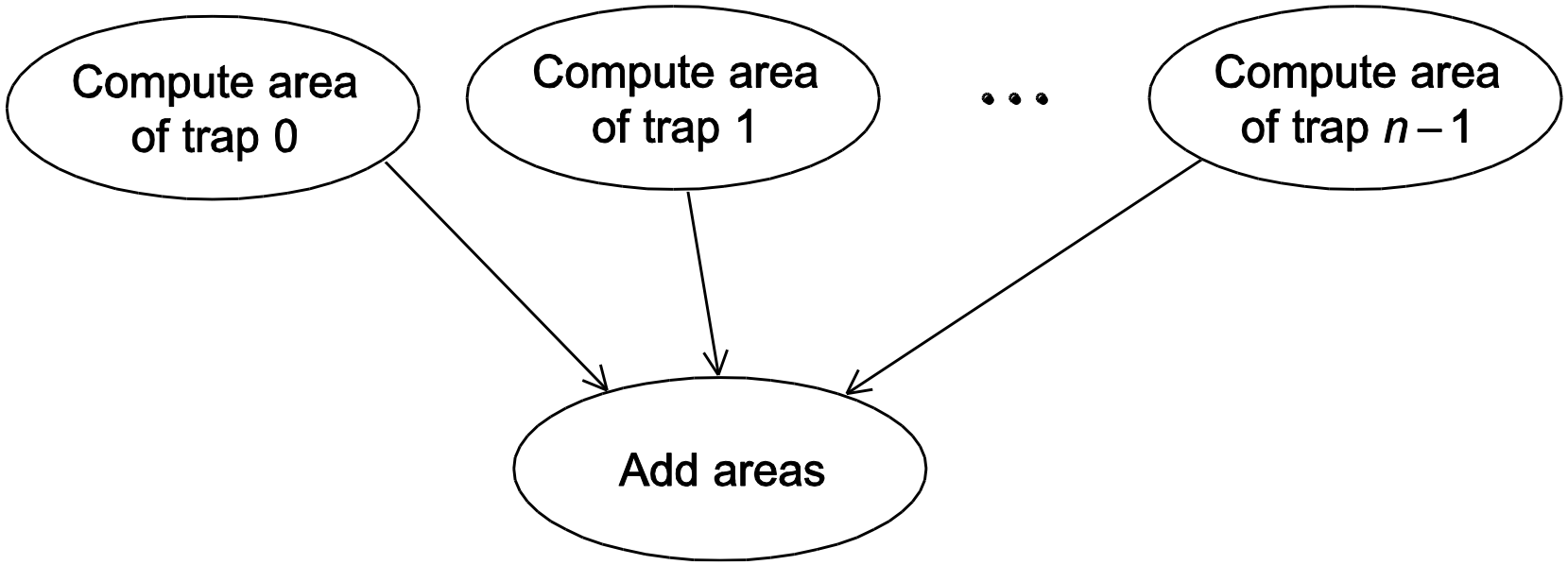
Sum of trapezoid areas $= h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$

```
/* 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.

Tasks and communications for Trapezoidal Rule



To which pattern does it look like?

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;
```

First version (part 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 (part 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 (part 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  */
```

Questions?

Input

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

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

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

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

Function for reading user input

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

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

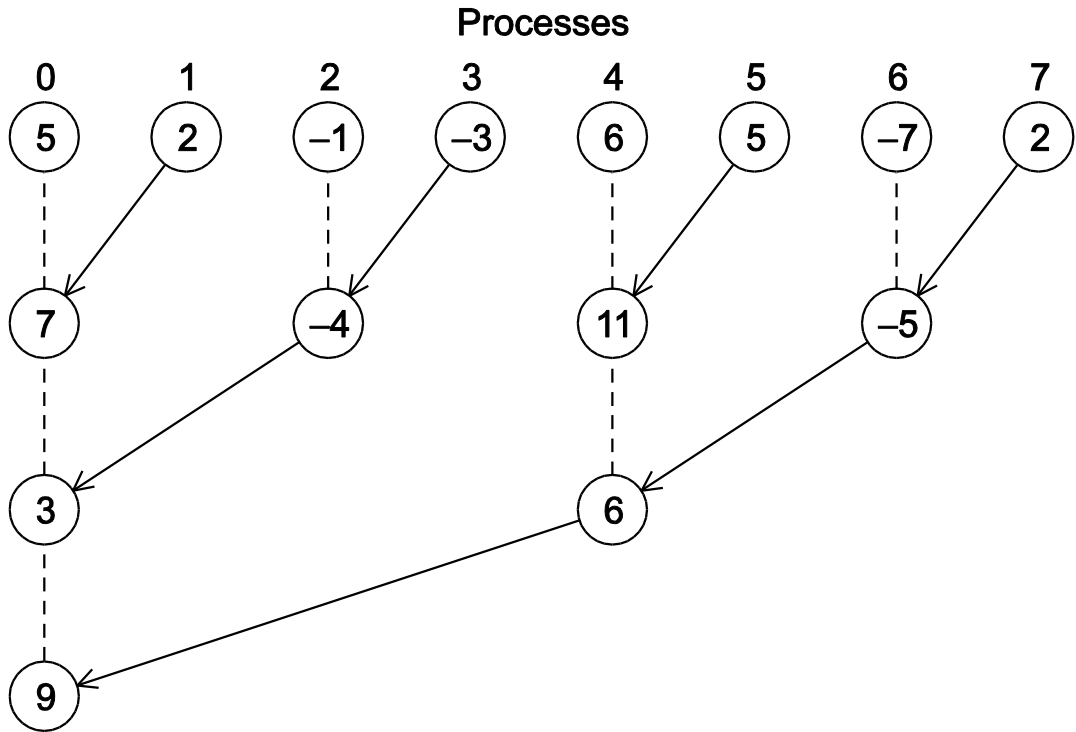
Issues with the trapezoidal rule implementation (1)

When doing the global sum, p-1 processes send their data to one process, which then computes all the sums. Unbalance! How long does it take?

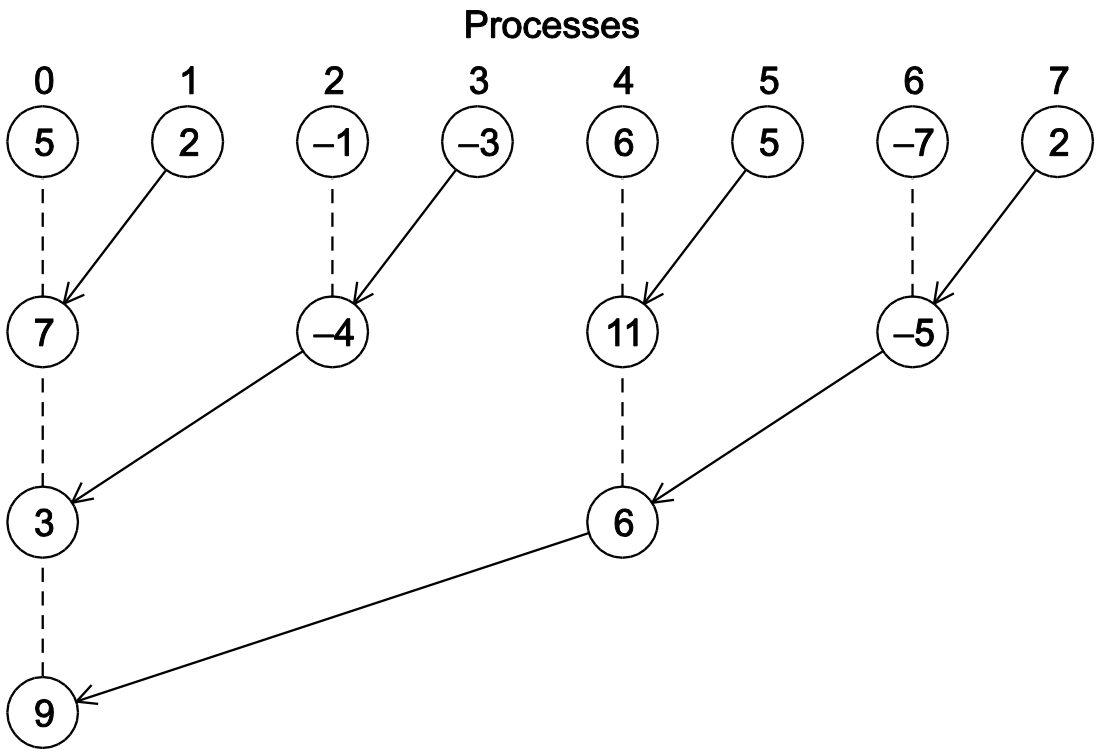
- For process 0: $(p-1)*(T_{sum} + T_{recv})$
- For all the other processes: T_{send}

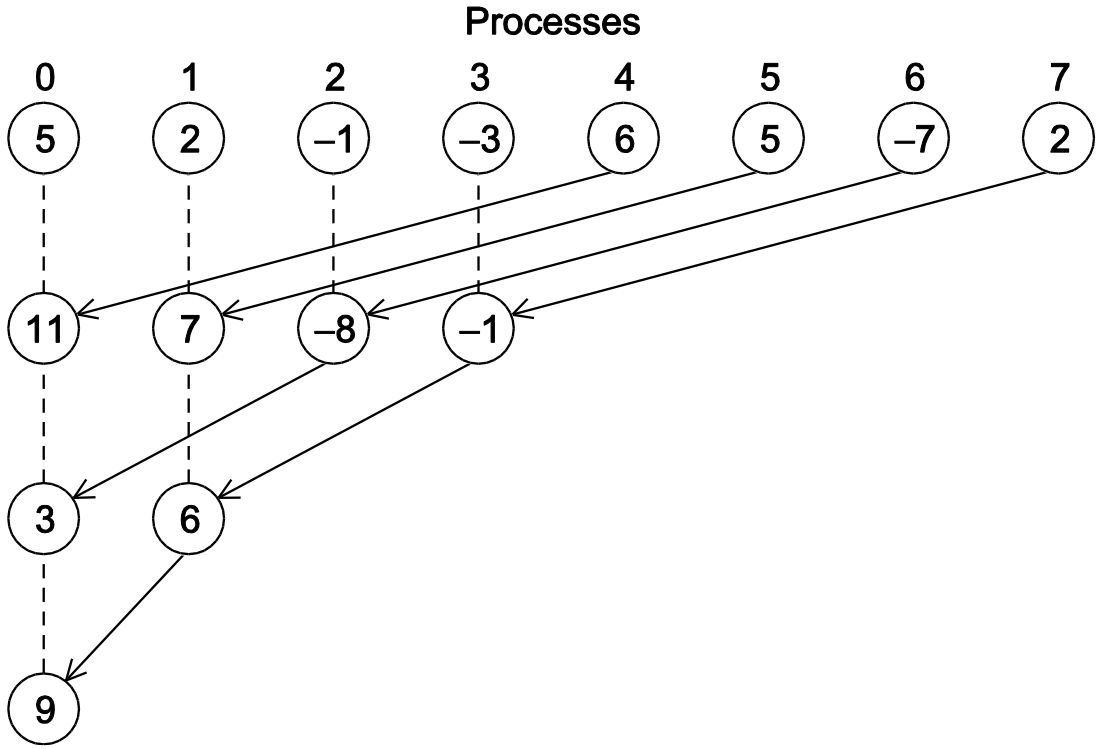
Alternative

- For process 0: $\log2(p)*(T_{sum} + T_{recv})$



Different valid trees

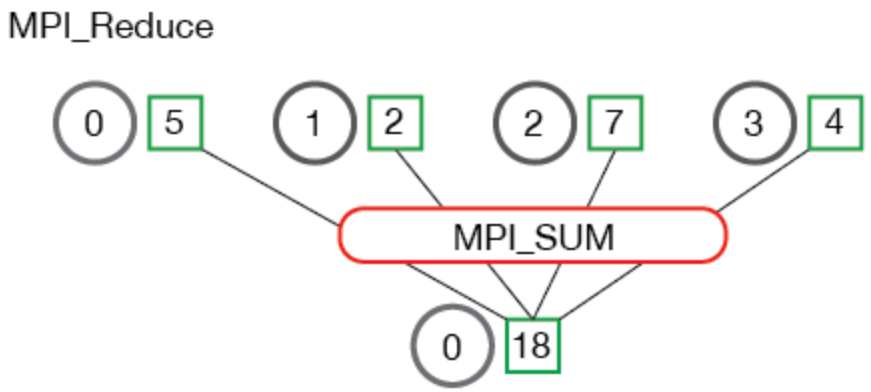


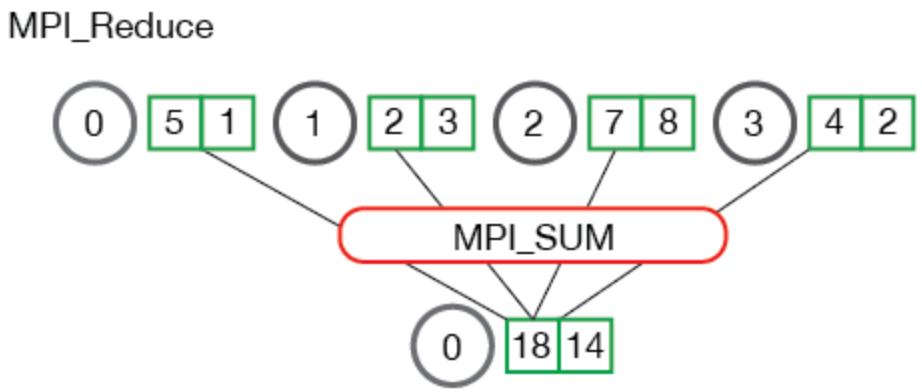


- The optimal way to compute a global sum depends on the number of processes, the size of the data, and the system we are running on (how many NICs, how the nodes are connected, etc…)
- Having a native way to express the global sum would simplify programming and improve performance

Collective Communication

MPI_Reduce


MPI_Reduce
0
5
1
2
2
7
3
4
MPI_SUM
0
18


MPI_Reduce
0
5
1
1
2
3
2
7
8
3
4
2
MPI_SUM
0
18
14

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

One call for all the processes

```c
int main(void) {
   int my_rank, comm_sz, n, local_n;
   double a, b, h, local_a, local_b;
   double local_int, total_int;

   MPI_Init(NULL, NULL);
   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;          /* h is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   /* Length of each process' interval of
    * integration = local_n*h.  So my interval
    * starts at: */
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   /* Add up the integrals calculated by each process */
   MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }

   /* Shut down MPI */
   MPI_Finalize();

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

MPI_Reduce Operators

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

Questions?

Caveats

- 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.
- 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.
- 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.
- 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.

Matching Example

All the following calls are done on MPI_COMM_WORLD, have 0 as destination, and MPI_SUM as operator

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

b = 1+2+1=4
d = 2+1+2=5

b = ???
d = ???

b = ???
d = ???

Questions?

Issues with the trapezoidal rule implementation (2)

```
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 */
```

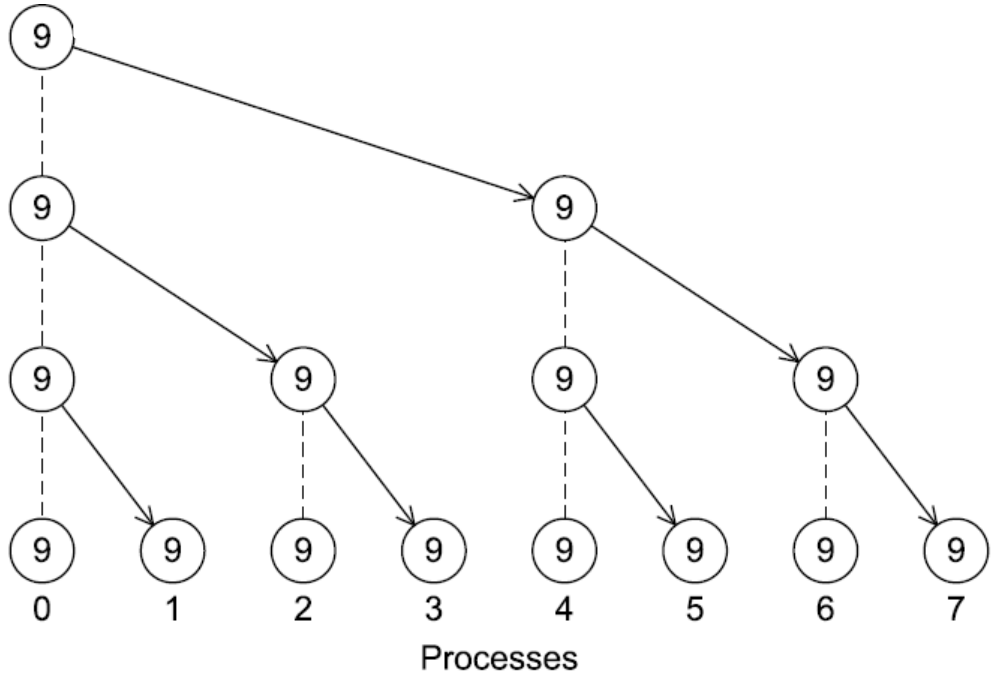
When doing the global sum, p-1 processes recv the data from one process. Unbalance! How long does it take?

For process 0: $(p-1)*(T_{send})$

For all the other processes: T_{recv}

Alternative

- For process 0: $\log2(p)*(T_{send})$



MPI_Bcast

- 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

```
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 */
```

Questions?

MPI_Allreduce

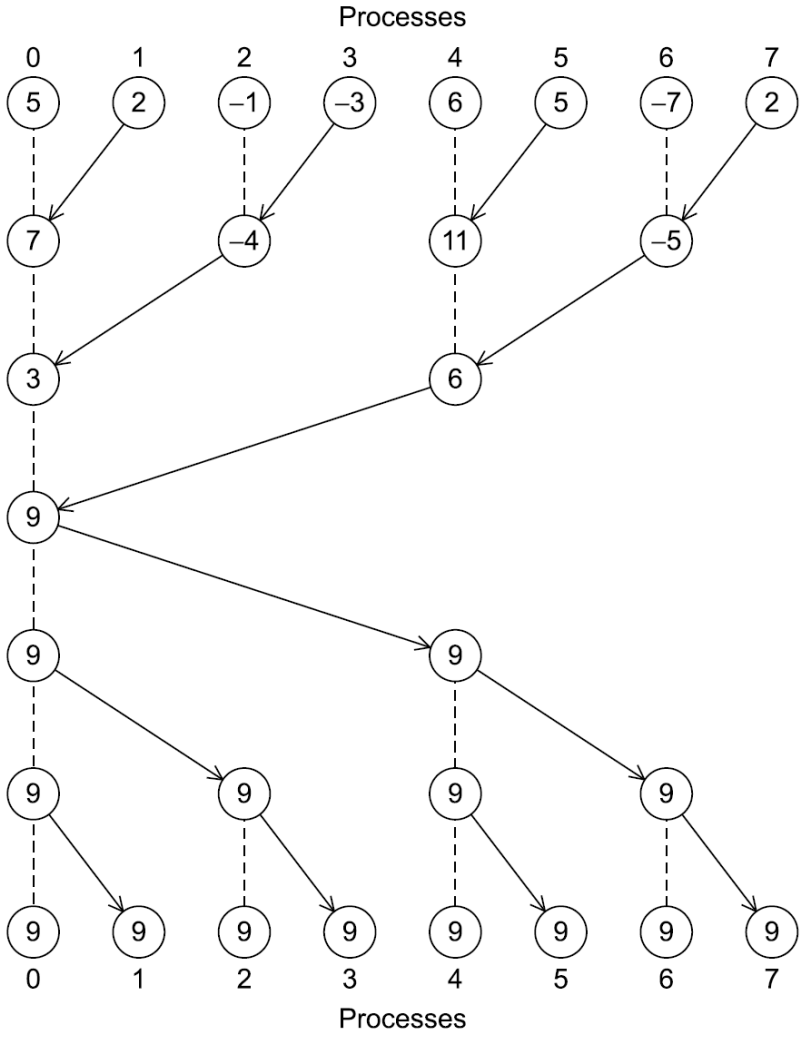
- Conceptually, an MPI_Reduce followed by MPI_Bcast (i.e., compute a global sum and distribute the result to all the processes)

```
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  */);
```

The argument list is identical to that for `MPI_Reduce`, except that there is no `dest_process` since all the processes should get the result.

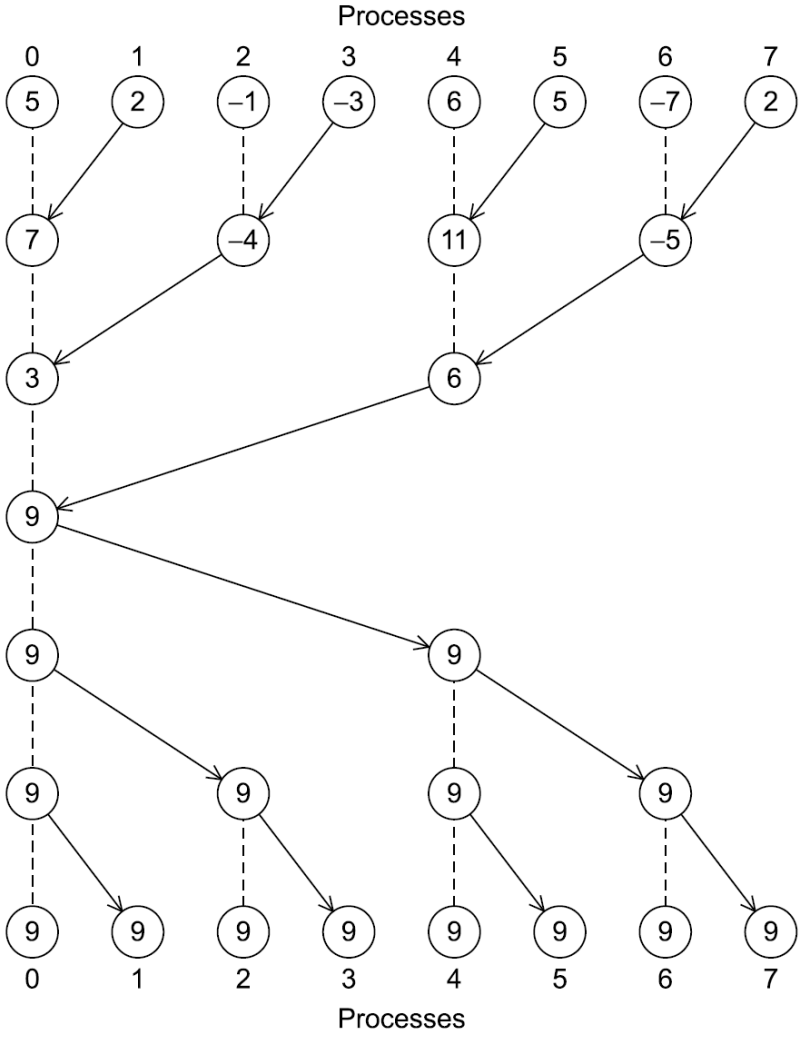
MPI_Allreduce

- Conceptually, an MPI_Reduce followed by MPI_Bcast (i.e., compute a global sum and distribute the result to all the processes)

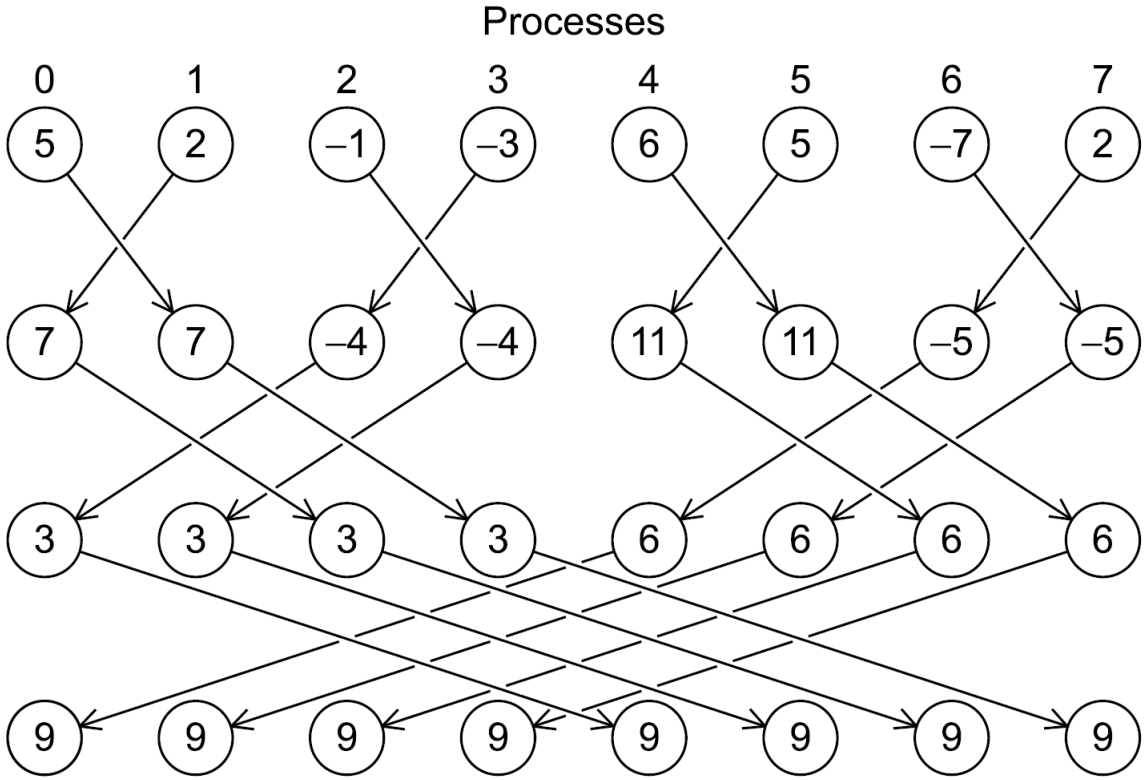


Is this the best way of doing it?

MPI_Allreduce



$T = 2*\log2(p)*T_{send}$



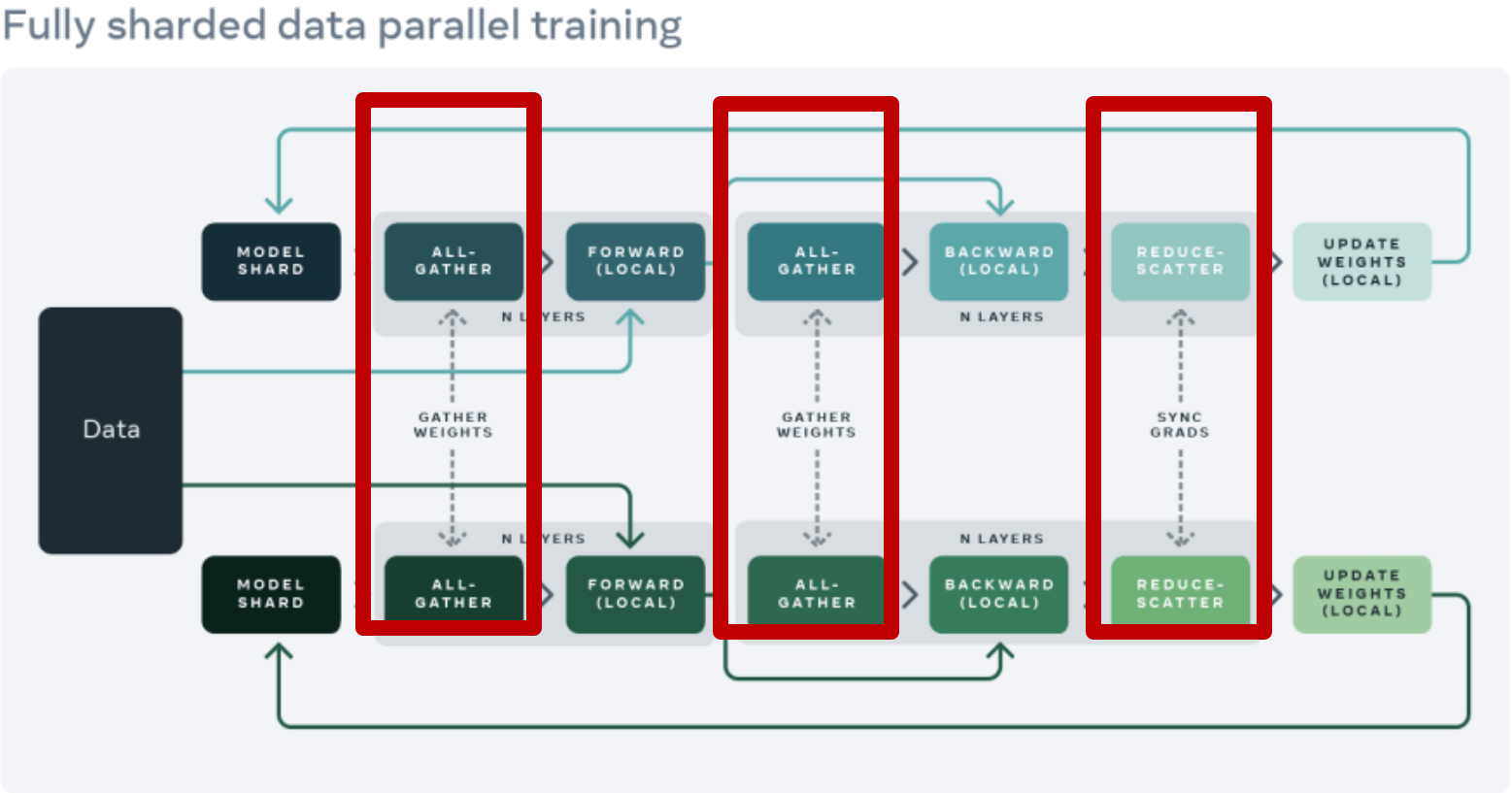
This is also known as **butterfly** pattern
(sometimes as recursive distance doubling)

$T = \log2(p)*(T_{send})$ **(Assuming send and recv happen at the same time)**

2x faster (other algos might be better depending on the data size)

Relevance of collective algorithms

- Widely used in large-scale parallel applications from many domains
- Account for a large fraction of the total runtime
- Highly relevant for distributed training of deep-learning models, e.g., Meta's FSDP training system:



https://engineering.fb.com/2021/07/15/open-source/fsdp/

Relevance of collective algorithms

- That's the reason why all the big players are designing their own collective communication library. E.g.,
  - NCCL (NVIDIA)
  - RCCL (AMD)
  - OneCCL (Intel)
  - MSCCL (Microsoft)
  - ….
- Given a collective (e.g., MPI_Reduce), how to select the best algorithm?
  - Automatically through heuristic
  - Manually
  - MPI implementations such as Open MPI do not make assumption on the underlying hardware, *CCL does
- Active research area, both from algorithmic and implementations standpoints

Exercises

Exercises

3.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

i.e., number in circle : pi = total number of tosses : 4

Remember that if origin (0,0), the circle must respect the condition $x^2+y^2=r^2$

We can use this formula to estimate the value of with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses). Write an MPI program that uses a Monte Carlo method to estimate π.

Exercises

3.2. Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide *n*. (You can still assume that $n \geq comm\_sz$)

3.4. Modify the program that just prints a line of output from each process (*mpi_output.c*) so that the output is printed in process rank order: process Os output first, then process 1s, and so on.

Exercises

3.1. Use MPI to implement the histogram program. Have process 0 read in the input data and distribute it among the processes. Also have process 0 print out the histogram.

Exercises

3.3. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which *comm_sz* is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any *comm_sz*

3.9. Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that *n*, the order of the vectors, is evenly divisible by comm sz.

3.13. MPI Scatter and MPI Gather have the limitation that each process must send or receive the same number of data items. When this is not the case, we must use the MPI functions MPI Gatherv and MPI Scatterv. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when *n* isn't evenly divisible by comm sz.