

# Distributed Memory Programming with Message-Passing

---

Pacheco's book Chapter 3

T. Yang, CS240A 2016

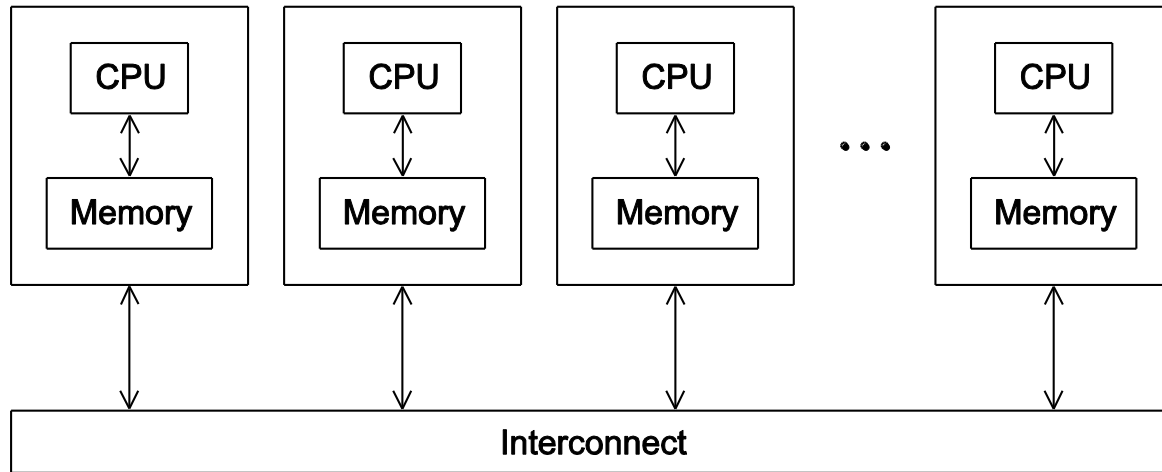
Part of slides from the text book and B. Gropp

# Outline

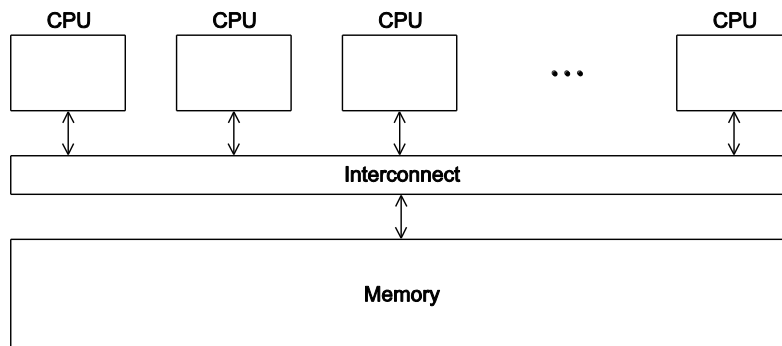
---

- **An overview of MPI programming**
  - Six MPI functions and hello sample
  - How to compile/run
- **More on send/receive communication**
- **Parallelizing numerical integration with MPI**

# Mainly for distributed memory systems



**Not targeted for shared memory machines. But can work**



# Message Passing Libraries

---

- MPI, Message Passing Interface, now the industry standard, for C/C++ and other languages
- **Running as a set of processes. No shared variables**
- **All communication, synchronization require subroutine calls**
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?
  - Communication
    - point-to-point: Send and Receive
    - Collectives such as broadcast
  - Synchronization
    - Barrier

# MPI Implementations & References

---

- **The Standard itself (MPI-2, MPI-3):**
  - at <http://www.mpi-forum.org>
- **Implementation for Linux/Windows**
  - Vendor specific implementation
  - MPICH
  - Open MPI
- **Other information on Web:**
  - [http://en.wikipedia.org/wiki/Message\\_Passing\\_Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface)
  - <http://www.mcs.anl.gov/mpi> MPI talks and tutorials, a FAQ, other MPI pages

# MPI is Simple

---

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECV`
- To measure time: `MPI_Wtime()`

# Mpi\_hello (C)

---

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d!\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

# Compilation

---

*wrapper script to compile*

*source file*

`mpicc -O -o mpi_hello mpi_hello.c`

*Mix with openmp*

`mpicc -O -o mpi_hello mpi_hello.c -fopenmp`



# Execution with mpirun or mpiexec

---

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

---

`mpirun -n 1 ./mpi_hello`

 *run with 1 process*

`mpirun -n 4 ./mpi_hello`

 *run with 4 processes*

# Execution

```
mpirun -n 1 ./mpi_hello
```

```
I am 0 of 1 !
```

```
mpirun -n 4 ./mpi_hello
```

```
I am 0 of 4 !
```

```
I am 1 of 4 !
```

```
I am 2 of 4 !
```

```
I am 3 of 4 !
```

# Running an MPI job at Comet

---

```
#!/bin/bash
#SBATCH --job-name="hellompi"
#SBATCH --output="hellompi.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=24
#SBATCH --export=ALL
#SBATCH -t 01:30:00
#This job runs with 2 nodes, 24 cores per node for a total of 48 cores.
#ibrun in verbose mode will give binding detail
```

```
ibrun -v ../hello_mpi
```

# MPI Programs

---

- **Written in C/C++.**
  - Has main.
  - Uses `stdio.h`, `string.h`, etc.
- **Need to add `mpi.h` header file.**
- **Identifiers defined by MPI start with “MPI\_”.**
- **First letter following underscore is uppercase.**
  - For function names and MPI-defined types.
  - Helps to avoid confusion.
- **MPI functions return error codes or `MPI_SUCCESS`**

# MPI Components

- **MPI\_Init**

- Tells MPI to do all the necessary setup.

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

- **MPI\_Finalize**

- Tells MPI we're done, so clean up anything allocated for this program.

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

# Basic Outline

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

. . .
int main(int argc, char* argv[]) {
    . . .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);

    . . .
    MPI_Finalize();
    /* No MPI calls after this */

    . . .
    return 0;
}
```

# Basic Concepts: Communicator

---

- Processes can be collected into groups
  - Communicator
  - Each message is sent & received in the same communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called `MPI_COMM_WORLD`

# Communicators



```
int MPI_Comm_size(  
    MPI_Comm comm      /* in */,  
    int* comm_sz_p    /* out */);
```

*number of processes in the communicator*

```
int MPI_Comm_rank(  
    MPI_Comm comm      /* in */,  
    int* my_rank_p    /* out */);
```

*my rank*  
*(the process making this call)*

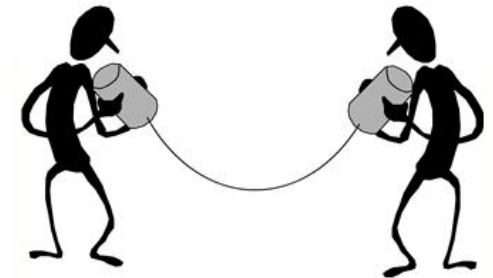


# Basic Send

```
int MPI_Send(  
  
    void*      msg_buf_p      /* in */,  
    int       msg_size       /* in */,  
    MPI_Datatype msg_type     /* in */,  
    int       dest           /* in */,  
    int       tag            /* in */,  
    MPI_Comm  communicator   /* in */);
```

- **Things specified:**

- How will “data” be described?
- How will processes be identified?
- How will the receiver recognize/screen messages?

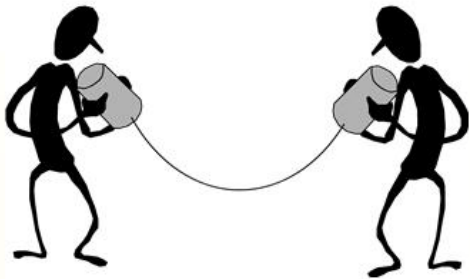


# Data types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

# Basic Receive: Block until a matching message is received

```
int MPI_Recv(  
    void*          msg_buf_p      /* out */,  
    int           buf_size       /* in */,  
    MPI_Datatype  buf_type       /* in */,  
    int           source         /* in */,  
    int           tag            /* in */,  
    MPI_Comm      communicator    /* in */,  
    MPI_Status*   status_p       /* out */);
```



## • Things that need specifying:

- Where to receive data
- How will the receiver recognize/screen messages?
- What is the actual message received

# Message matching

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

*MPI\_Send*  
*dest*

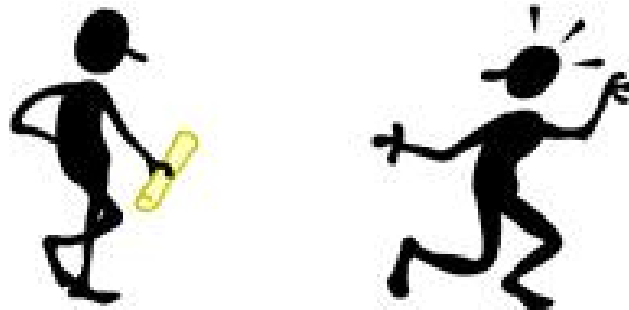


*MPI\_Recv*  
*src*

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

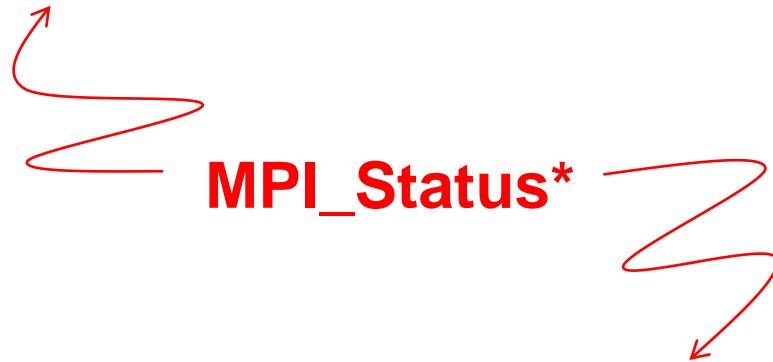
# Receiving messages without knowing

- **A receiver can get a message without knowing:**
  - the amount of data in the message,
  - the sender of the message,
    - Specify the source as `MPI_ANY_SOURCE`
  - or the tag of the message.
    - Specify the tag as `MPI_ANY_TAG`



# Status argument: who sent me and what tag is?

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



- Who sent me*
- What tag is*
- Error code*
- Actual message length*

# Retrieving Further Information from status argument in C

- **Status is a data structure allocated in the user's program.**
- **In C:**

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

# Retrieving Further Information in C++

- **Status** is a data structure allocated in the user's program.
- **In C++:**

```
int recvd_tag, recvd_from, recvd_count;
MPI::Status status;
Comm.Recv(..., MPI::ANY_SOURCE, MPI::ANY_TAG, ...,
          status )

recvd_tag   = status.Get_tag();
recvd_from  = status.Get_source();
recvd_count = status.Get_count( datatype );
```

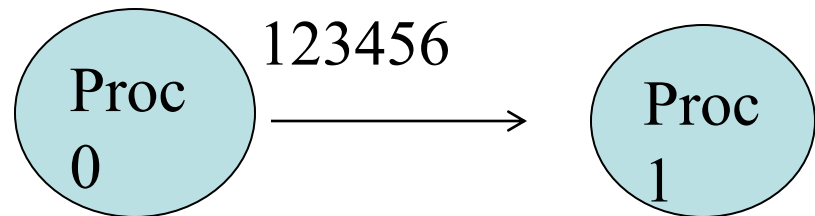


# MPI Example: Simple send/recv

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argv, &argc);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Process 0 sends and Process 1 receives */
    if (rank == 0) {
        buf = 123456;
        MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 &status );
        printf( "Received %d\n", buf );
    }

    MPI_Finalize();
    return 0;
}
```

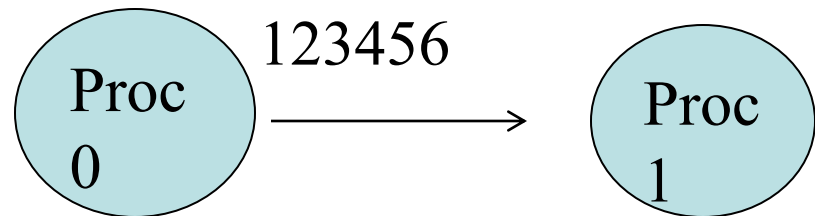


# MPI Send/Receive Example with C++

```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI::Init(argv, argc);
    rank = MPI::COMM_WORLD.Get_rank();

    // Process 0 sends and Process 1 receives
    if (rank == 0) {
        buf = 123456;
        MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
    }
    else if (rank == 1) {
        MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
        std::cout << "Received " << buf << "\n";
    }

    MPI::Finalize();
    return 0;
}
```



Slide source: Bill Gropp, ANL

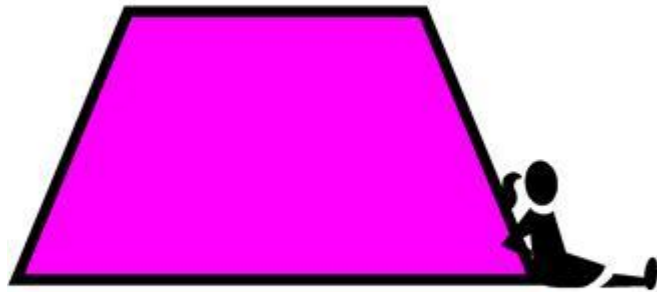
## ***MPI\_Wtime()***

---

- **Returns the current time with a double float.**
- **To time a program segment**
  - *Start time = MPI\_Wtime()*
  - *End time = MPI\_Wtime()*
  - Time spent is *end\_time – start\_time.*

# Example of using MPI\_Wtime()

```
#include<stdio.h>
#include<mpi.h>
main(int argc, char **argv){
    int size, node;          double start, end;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    start = MPI_Wtime();
    if(node==0) {
        printf(" Hello From Master. Time = %lf \n", MPI_Wtime() -
            start);
    }
    else {
        printf("Hello From Slave #%d %lf \n", node, (MPI_Wtime()
            - start));
    }
    MPI_Finalize();
}
```



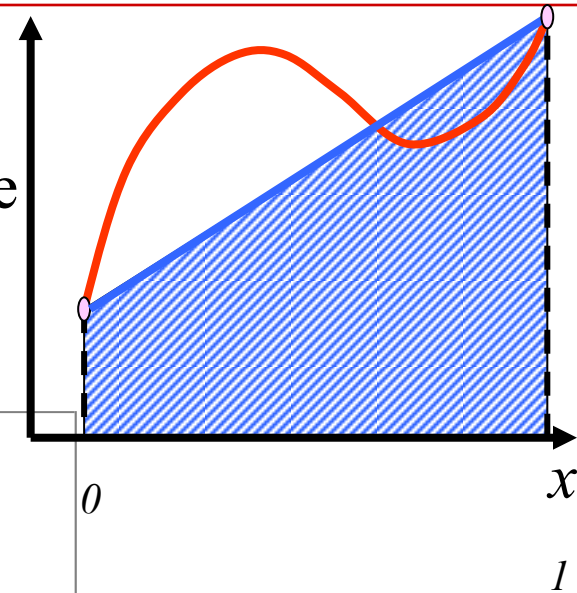
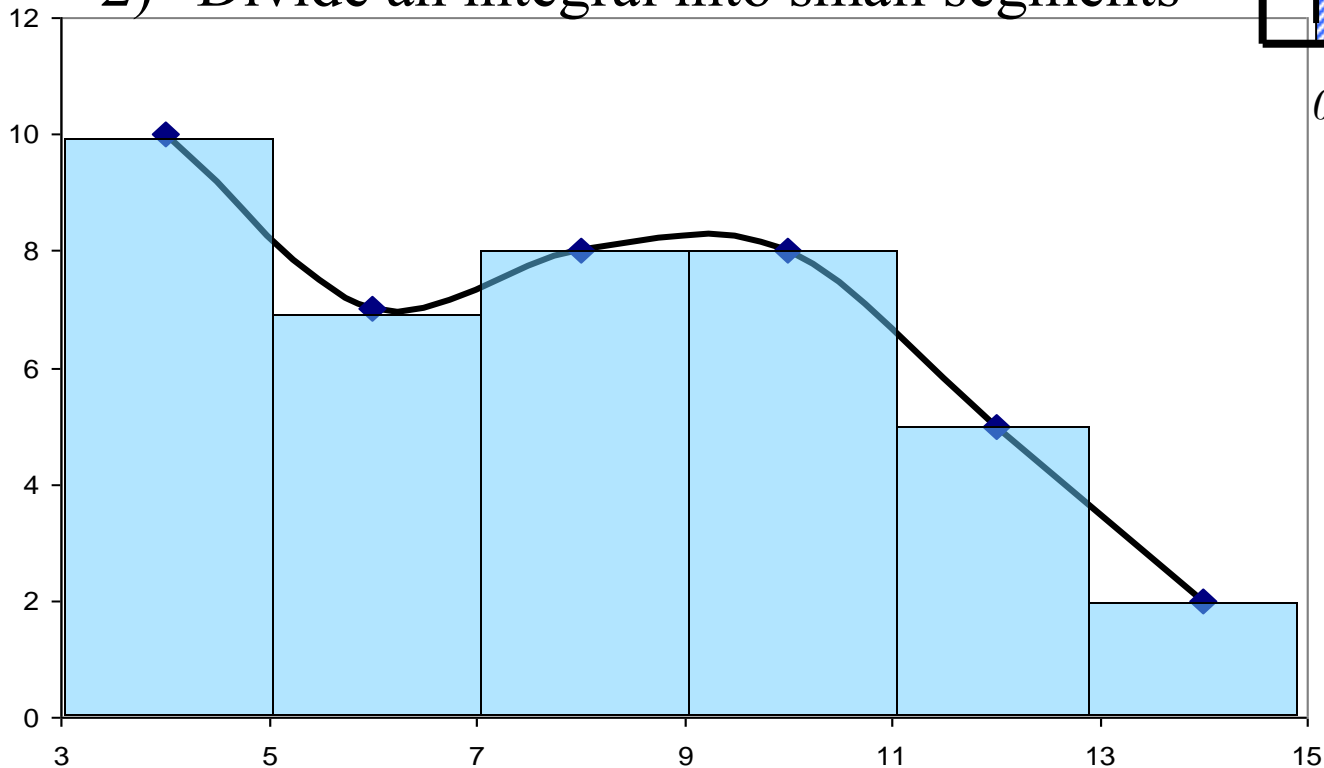
## **MPI Example: Numerical Integration With Trapezoidal Rule**

**PACHECO'S BOOK p. 94-101**

# Approximation of Numerical Integration

## Two ideas

- 1) Use a simple function to approximate the integral area.
- 2) Divide an integral into small segments

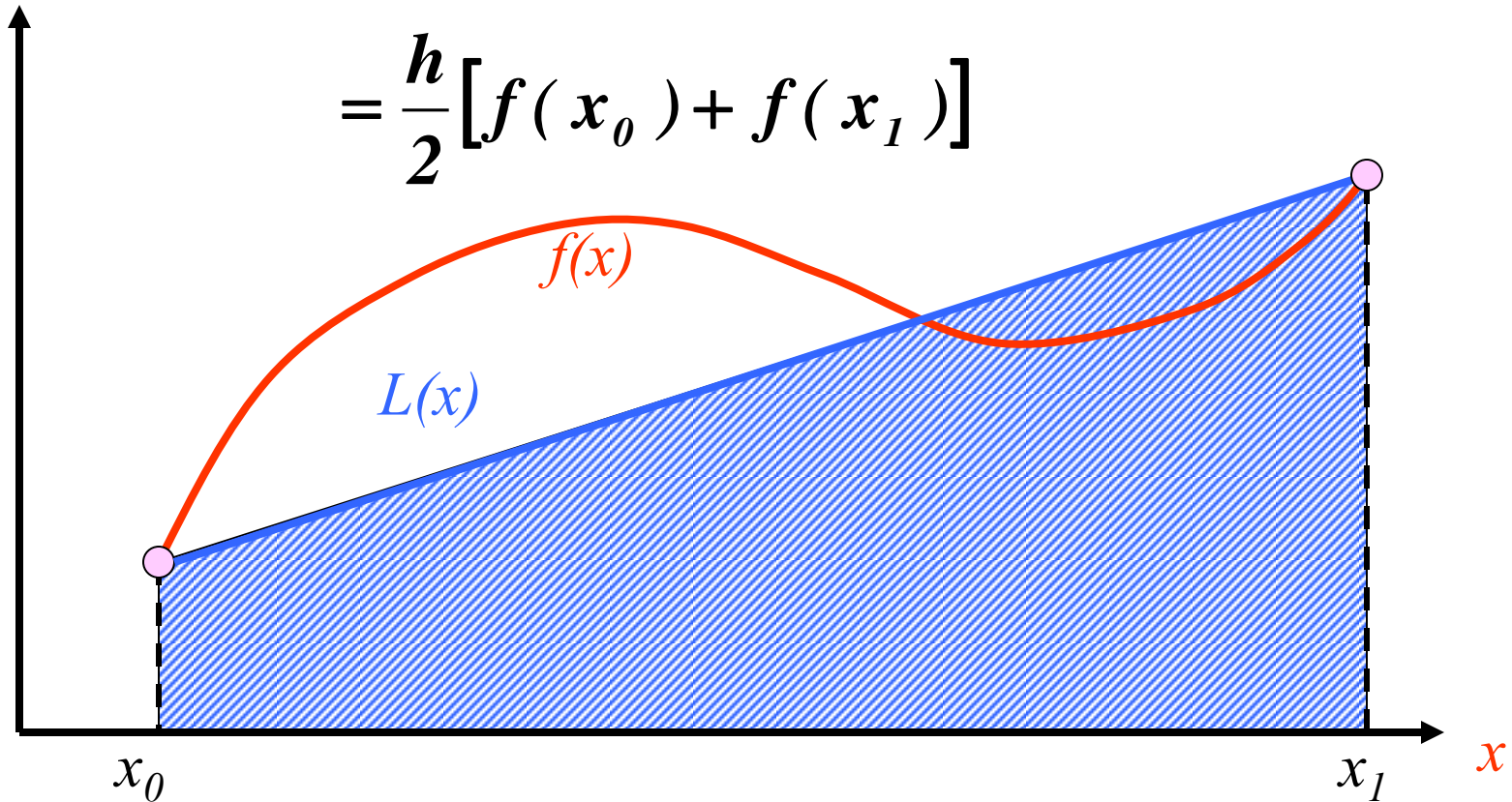


# Trapezoid Rule

- Straight-line approximation

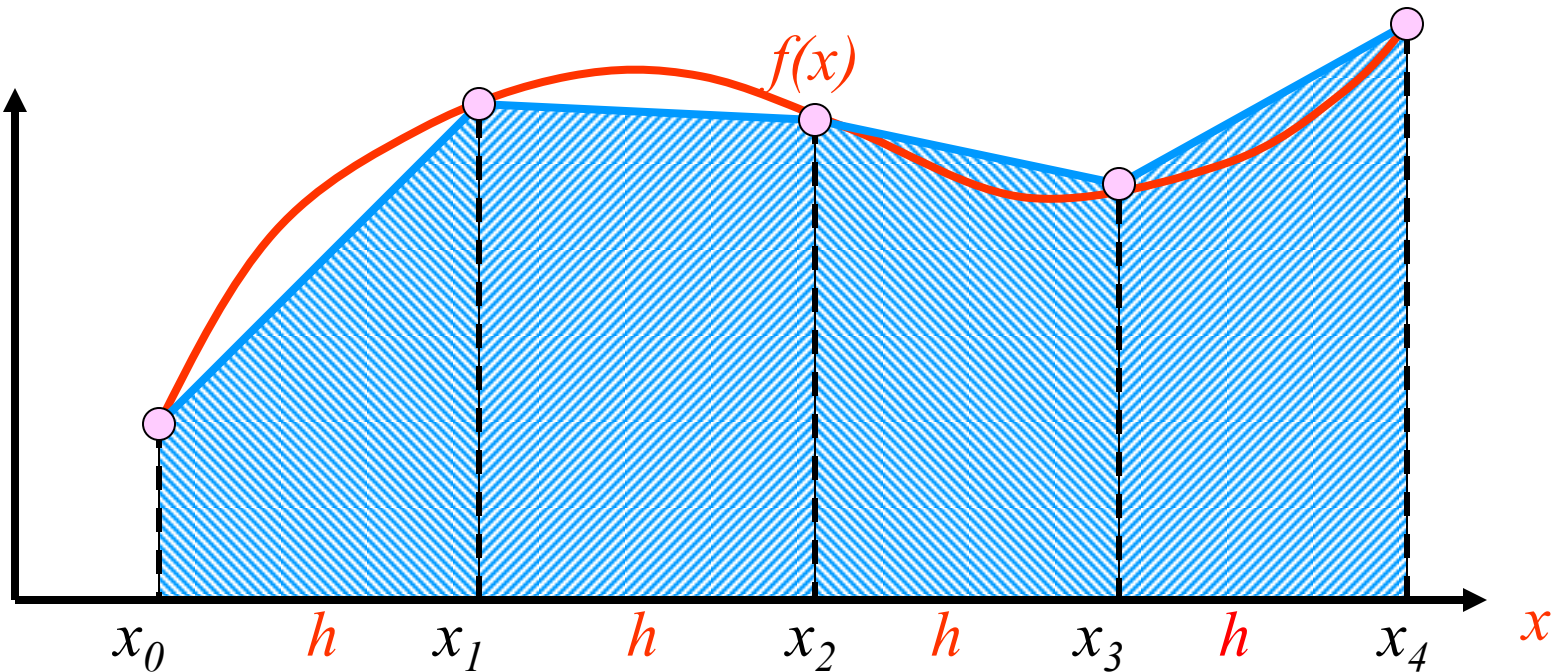
$$\int_a^b f(x) dx \approx \sum_{i=0}^1 c_i f(x_i) = c_0 f(x_0) + c_1 f(x_1)$$

$$= \frac{h}{2} [f(x_0) + f(x_1)]$$



# Composite Trapezoid Rule

$$\begin{aligned}\int_a^b f(x)dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx \\ &= \frac{h}{2} [f(x_0) + f(x_1)] + \frac{h}{2} [f(x_1) + f(x_2)] + \cdots + \frac{h}{2} [f(x_{n-1}) + f(x_n)] \\ &= \frac{h}{2} [f(x_0) + 2f(x_1) + \cdots + 2f(x_i) + \cdots + 2f(x_{n-1}) + f(x_n)]\end{aligned}$$



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

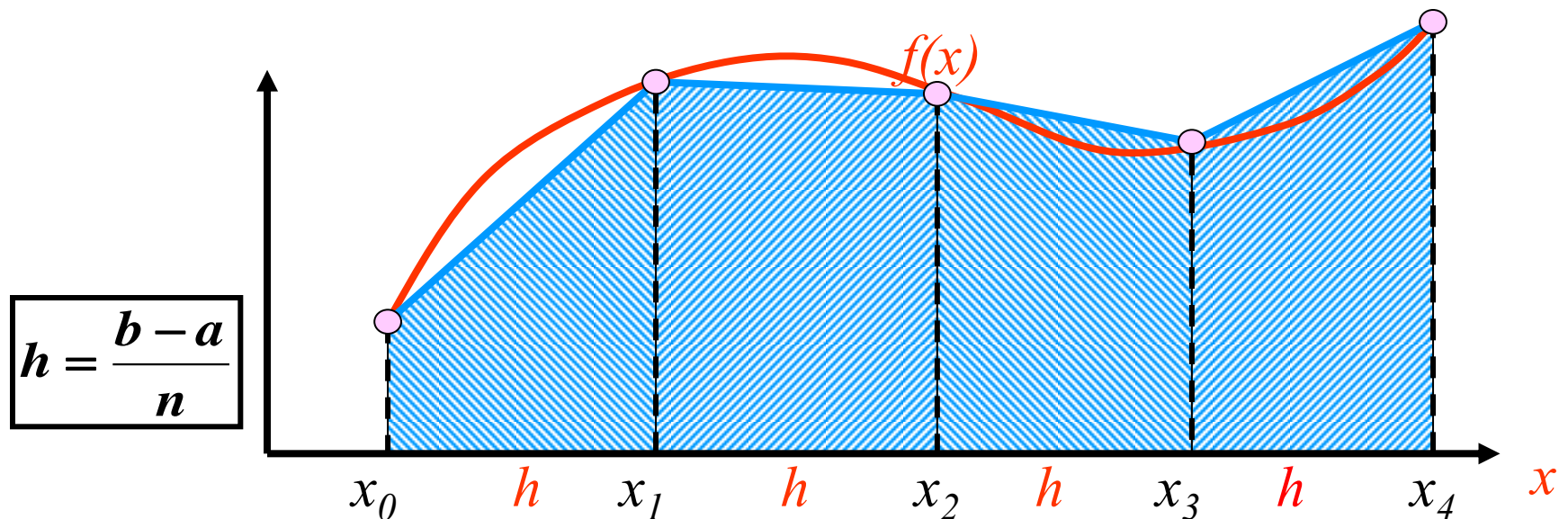


# Implementing Composite Trapezoidal Rule

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

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

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



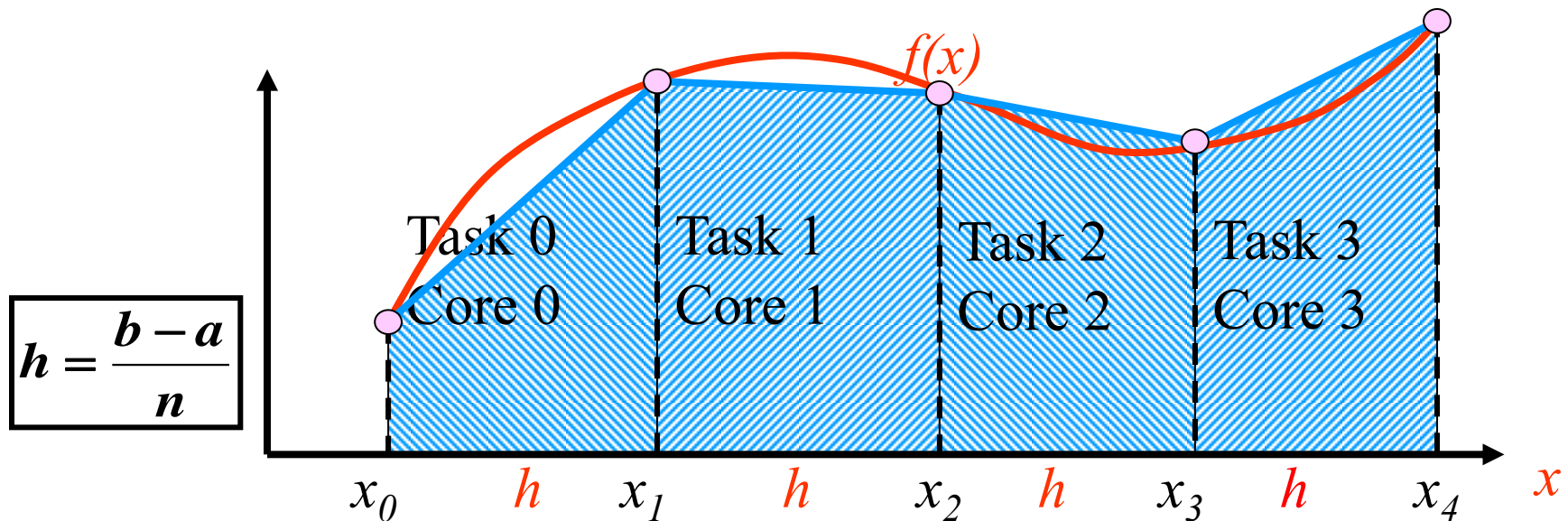
## Pseudo-code for a serial program

---

```
/* Input: a, b, n */  
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 0; i <= n-1; i++) {  
    x_i = a + i*h;  
    approx += f(x_i);  
}  
approx = h*approx;
```

# Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.



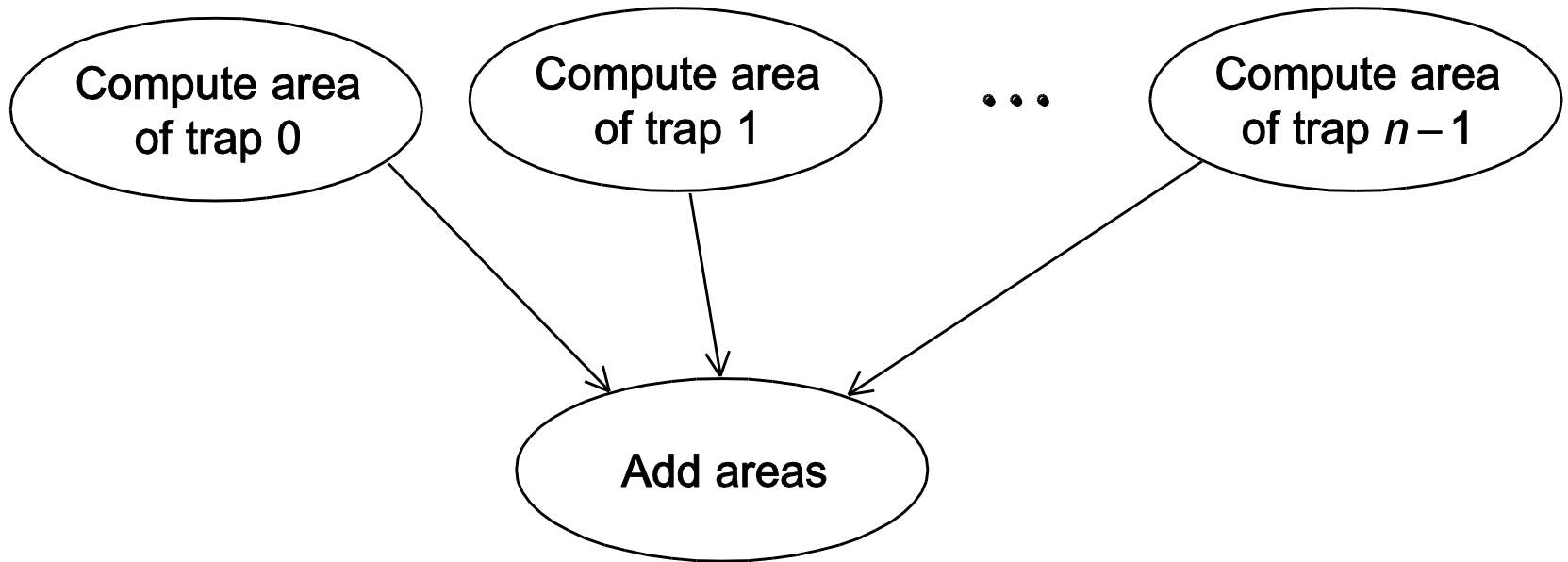
# Parallel pseudo-code

```
1   Get a, b, n;
2   h = (b-a)/n;
3   local_n = n/comm_sz;
4   local_a = a + my_rank*local_n*h;
5   local_b = local_a + local_n*h;
6   local_integral = Trap(local_a, local_b, local_n, h);
7   if (my_rank != 0)
8       Send local_integral to process 0;
9   else /* my_rank == 0 */
10      total_integral = local_integral;
11      for (proc = 1; proc < comm_sz; proc++) {
12          Receive local_integral from proc;
13          total_integral += local_integral;
14      }
15  }
16  if (my_rank == 0)
17      print result;
```

Compute the local area

Summation of local values

# Tasks and communications for Trapezoidal Rule



# First version (1)

```
1  int main(void) {
2      int my_rank, comm_sz, n = 1024, local_n;
3      double a = 0.0, b = 3.0, h, local_a, local_b;
4      double local_int, total_int;
5      int source;
6
7      MPI_Init(NULL, NULL);
8      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11     h = (b-a)/n;          /* h is the same for all processes */
12     local_n = n/comm_sz; /* So is the number of trapezoids */
13
14     local_a = a + my_rank*local_n*h;
15     local_b = local_a + local_n*h;
16     local_int = Trap(local_a, local_b, local_n, h);
17
18     if (my_rank != 0) {
19         MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20                 MPI_COMM_WORLD);
```

Use send/receive to sum

## First version (2)

```
21 } else {
22     total_int = local_int;
23     for (source = 1; source < comm_sz; source++) {
24         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26         total_int += local_int;
27     }
28 }
29
30 if (my_rank == 0) {
31     printf("With n = %d trapezoids, our estimate\n", n);
32     printf("of the integral from %f to %f = %.15e\n",
33         a, b, total_int);
34 }
35 MPI_Finalize();
36 return 0;
37 } /* main */
```

Use send/receive to sum

# First version: Trapezoidal Rule of local area

```
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 */
```



# I/O handling in trapezoidal program

- 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);
```

Process 0 inputs parameters

```
        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);  
    }
```

Broadcast parameters

```
} /* Get_input */
```