deep se
dependable evolvable pervasive software engineering group

Message Passing Interface

Eugenio Gianniti & Danilo Ardagna

Politecnico di Milano
name.lastname@polimi.it

06/12/2024

POLITECNICO
DI MILANO

Content

- Parallel Programming with MPI
- Preliminaries: Compiling C++ code and passing parameters to C++ programs
- Point to Point Communication
- Collective Communication

PARALLEL PROGRAMMING WITH MPI

Or how to pull off a parallel “Hello, world!”

What is MPI?

- MPI is an **interface specification**
  - Basically, a document stating the functionality that vendors should provide and users can rely upon
- The goal is a **portable**, flexible, **efficient**, and **practical** message passing **interface standard**
- The standard defines both a Fortran and a C specification and some more modern languages as Python
- Many alternative implementation exist
  - Our focus is on **Open MPI**

Programming Model

- In MPI all parallelism is **explicit**
  - Identifying what should be parallelized and how is on programmers
- MPI was designed for **distributed memory** architectures, even if the implementations currently support any common parallel architecture
- Hence, MPI virtually allows running your code on any parallel system

Message Passing Basic Idea

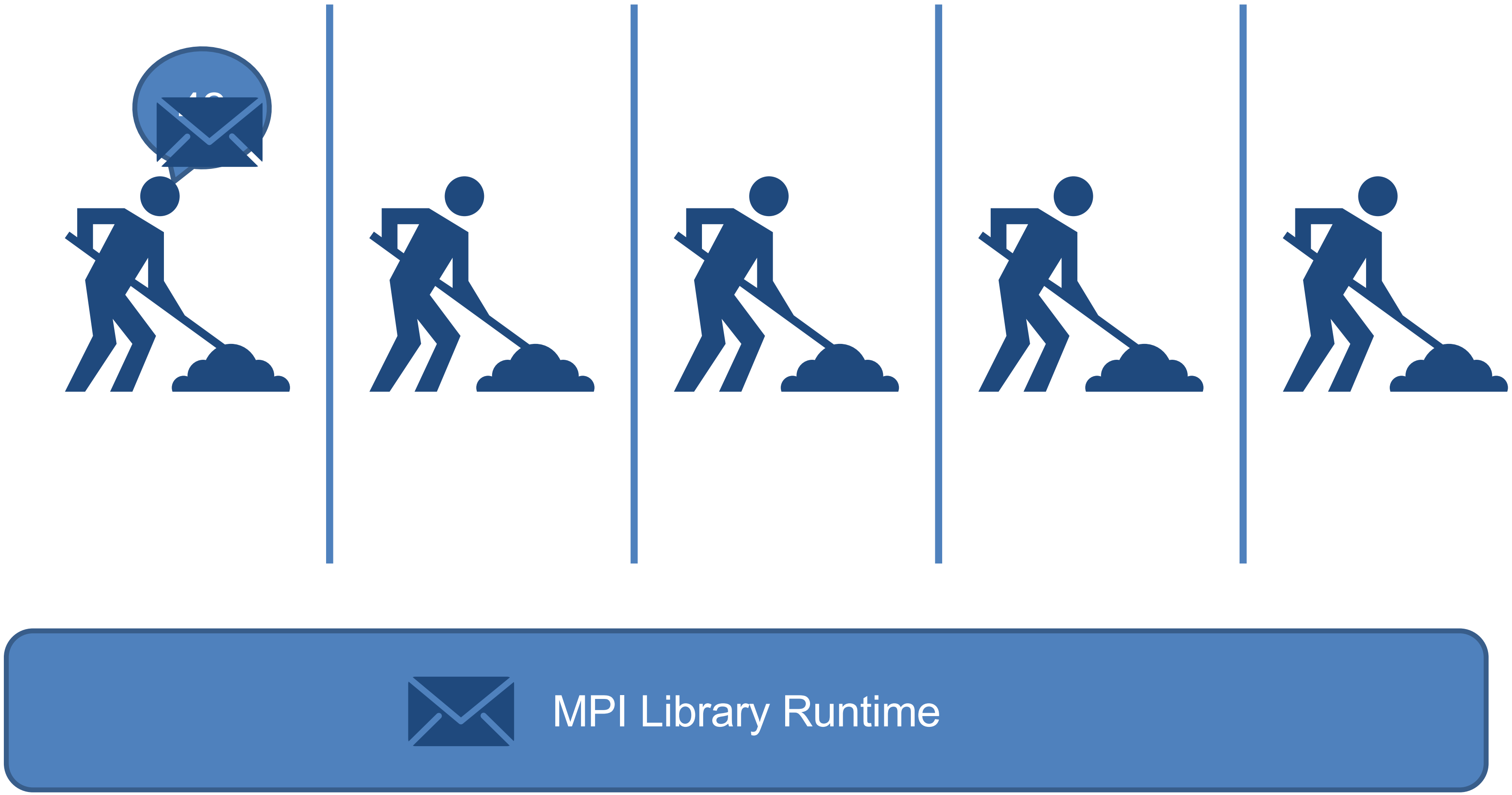
- In message-passing programs, a program running on one core is usually called a **process**

$$\text{process 1} \underset{\text{receive}}{\overset{\text{send}}{\rightleftarrows}} \text{process 2}$$

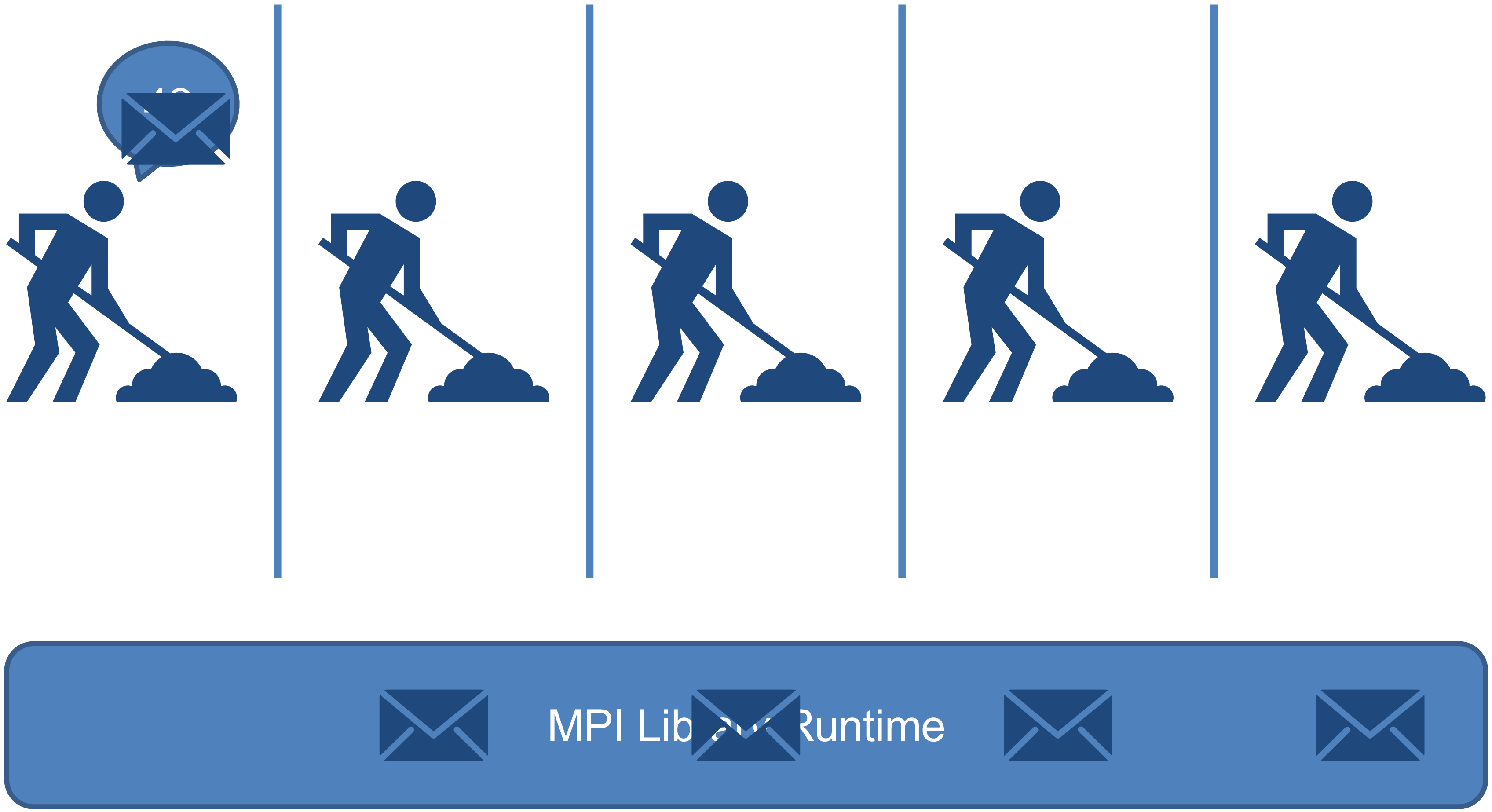
- Two processes can communicate by calling functions:
  - one process calls a *send* function
  - the other calls a *receive* function

- MPI supports also **global** communication functions that can involve more than two processes
  - These functions are called **collective** communications

MPI in Pictures – Send & Receive



MPI in Pictures – Collective (broadcast)



PRELIMINARIES: PASSING PARAMETERS TO C++ PROGRAMS

Or how to exploit multi-cores system to have fun!

Passing Arguments to main

- It is possible to pass some values from the command line to C/C++ programs when they are executed: **command line arguments**
- This is important when you want to control your program from outside instead of hard coding those values inside the code
- The command line arguments are handled using main() function arguments:
  - **argc** refers to the number of arguments passed
  - **argv[]** is an array of pointers, which point to each argument passed to the program
  - Each argument is represented as a C char[], hence **argv[] type is `char *`**

cf commandline.cpp

Passing Arguments to main

- Let’s consider a simple example which checks if there is a single argument from the command line and takes action accordingly

```
#include <iostream>
int main( int argc, char *argv[] )
{
        if( argc == 2 ) {
                std::cout << "The argument supplied is" <<
                                argv[1] << std::endl;
        }
        else if( argc > 2 )
        {
                std::cout << "Too many arguments
                                supplied." << std::endl;
        }
        else
                std::cout << " One argument expected ." << std::endl;
}
```

Argument 1 is always the program's name. Therefore when we test argc==2, we indeed check for the presence of 1 additional argument that we provide.
(cf 2 slides below)

Compiling your C++ program

```
$ g++ --std=c++23 file_name [-o executable]
```

Default is a.out if not specified

for me it's C++20.

1)
```
$ g++ --std=c++23 commandline.cpp
```

cf next slide for 2nd step.

```
apc-2024@apc-2024-VM:~/Desktop/APC2024
→ APC2024 ls
2processes.cpp         commandline.cpp   hello.cpp
2processes_debug.cpp   cyclic_part.cc    trapezoidal_rule
collective.cpp         hello2.cpp        vector_sum
→ APC2024 g++ --std=c++23 commandline.cpp
→ APC2024 ls
2processes.cpp         collective.cpp    hello2.cpp         vector_sum
2processes_debug.cpp   commandline.cpp   hello.cpp
a.out                  cyclic_part.cc    trapezoidal_rule
→ APC2024
```

Executing your program at the command line

2)
```
$ ./a.out testing
The argument supplied is testing
```
1 arg supplied: OK.

```
$ ./a.out testing1 testing2
Too many arguments supplied
```
2 args supplied: too many.

```
$ ./a.out
One argument expected
```
no arg. supplied: not enough.

- It should be noted that:
  - argv[0] is a pointer to the name of the program itself
  - argv[1] is a pointer to the first command line argument supplied
  - *argv[argc - 1] is the last argument

- If no arguments are supplied, argc will be one, and if you pass one argument then argc is set at 2

MPI BASICS

Or how to split your programs' personality

Hello World!

```
#include <cstdio>

int main (int argc, char *argv[])
{
  printf ("Hello world!\n");
  return 0;
}
```

classical way.

Hello World!

cf hello.cpp

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

int main (int argc, char *argv[])
{
  MPI_Init (&argc, &argv);
  int rank, size;
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  printf ("Hello from process %d of %d\n", rank, size);
  MPI_Finalize ();

}
```

You don't compile as before for classical C++; cf next slide.

In parallel programming, it's common (one might say standard) for the processes to be identified by nonnegative integer *ranks*. So if there are *size* processes, the processes will have ranks 0,1,2,..., *size*-1

process = 1 prog. running on 1 core.

Compile & Run MPI CODE

- To compile MPI code you use:

1) `$ mpicxx -o exe file1.cc [file2.cc ...]`
   - All the flags are the same as with `g++`

- To run MPI executables you use:

2) `$ mpiexec -np 4 exe`
   - If you use MPI on a cluster, you should also provide a file listing all the involved nodes with `-machinefile /path/to_node_list`

! TO BE COMPILED ON THE VM.

What to Expect

- Remember to compile with `mpicxx`!
  - Otherwise the linking stage will fail with missing symbols

- The output will be a number of lines reading:

  `Hello from process 0 of 4`

  - The order is random

- How many lines are output depends on the `-np` flag to `mpiexec`

DEMO

What to Expect

- Remember to compile with `mpicxx`!

compile

RUN

```
apc-2024@apc-2024-VM:~/Desktop/APC2024
→ APC2024 mpicxx --std=c++23 -o helloMPI hello.cpp
→ APC2024 mpiexec -np 2 ./helloMPI
Hello from process 1 of 2
Hello from process 0 of 2
→ APC2024 mpiexec -np 4 -oversubscribe ./helloMPI
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 3 of 4
Hello from process 2 of 4
→ APC2024
```

MPI_Init and MPI_Finalize

cf code slide 16

- `MPI_Init` tells the MPI system to do all the necessary setup:
  - Allocate storage for message buffers and decide which process gets which rank
  - No other MPI functions should be called before the program calls `MPI_Init`

MPI_Init and MPI_Finalize

```
int MPI_Init(int* argc_p, char*** argv_p)
```

- The arguments, `argc_p` and `argv_p`, are pointers to the arguments to main, `argc` and `argv`
  - when our program doesn't use these arguments, we can just pass `nullptr` for both

- Like most MPI functions, `MPI_Init` returns an `int` error code
  - in most cases we'll ignore these error codes

MPI_Init and MPI_Finalize

- `MPI_Finalize` tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed

```
int MPI_Finalize(void)
```

- In general, no MPI functions should be called after the call to `MPI_Finalize`

MPI programs general structure

```
...
#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;
}
```

POINT TO POINT COMMUNICATION

Or how to deliver postcards to your friends

Communicators

- MPI processes can be addressed via **communicators**
- A **communicator** is a collection of processes that can send messages to each other
- The standard provides mechanisms for defining your own
- One is predefined and collects each and every process created when launching the program: `MPI_COMM_WORLD`
- **Point to point** means that you explicitly state which among the communicator's processes you want to reach

Ranks and Size

- A communicator **size** is the number of processes it collects and allows to reach
- Every process is identified within a communicator by means of a **rank**, a unique integer in [0, size)

communicator

```
int MPI_Comm_size (MPI_Comm comm, int *size)

int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

$\in [\![0, size-1]\!]$

- For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, `MPI_Comm`
- `MPI_Comm_size` returns in its second argument the number of processes in the communicator
- `MPI_Comm_rank` returns in its second argument the calling process rank in the communicator

A Sorted "Hello, World!" — The way to go for std::cout

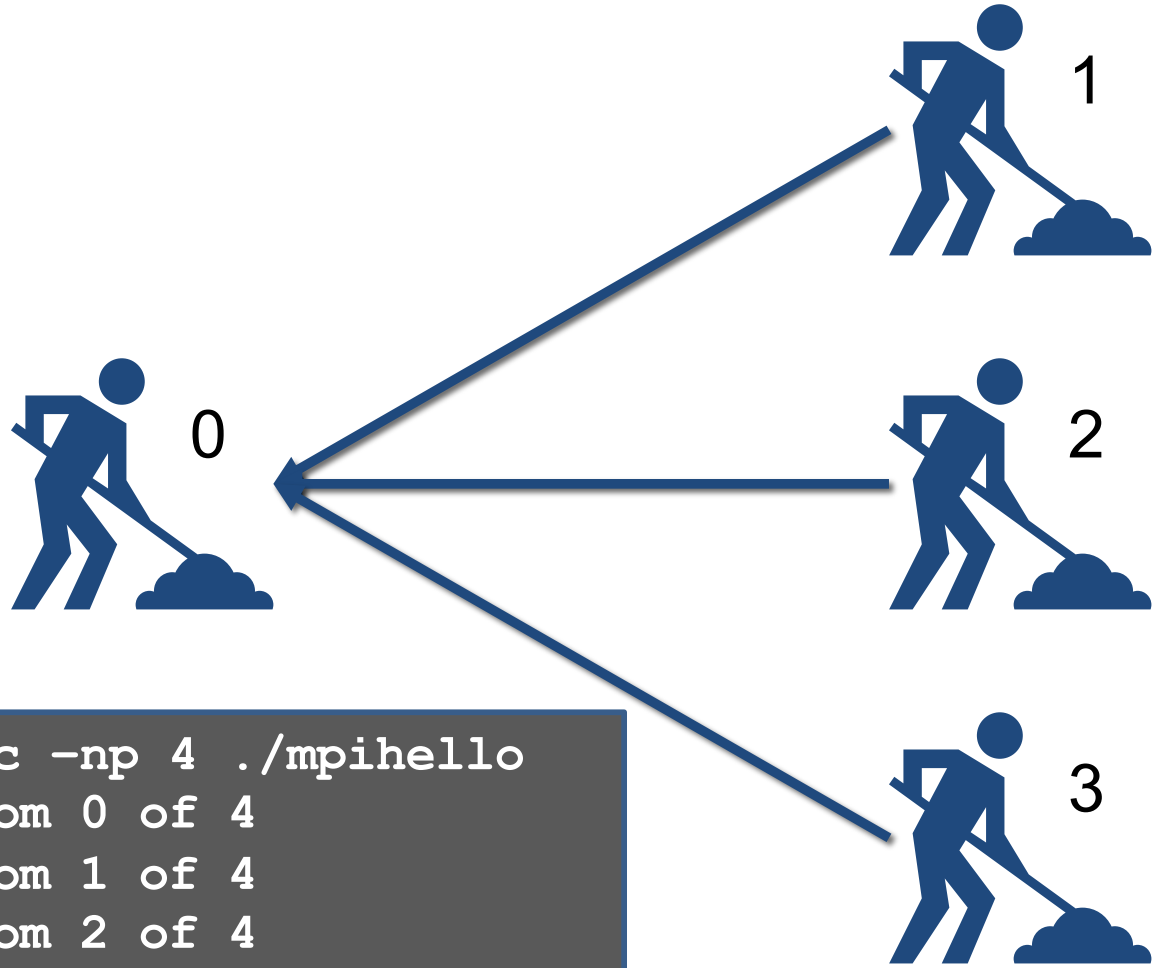
```
// Assumption we use up to 10 processes, no more!
constexpr unsigned max_string = 18;
std::ostringstream builder;
builder << "Hello from " << rank << " of " << size;
std::string message (builder.str ());
if (rank > 0)
  MPI_Send (&message[0], max_string , MPI_CHAR,
            0, 0, MPI_COMM_WORLD);
else
  {
    std::cout << message << std::endl;
    for (int r = 1; r < size; ++r)
      {
        MPI_Recv (&message[0], max_string , MPI_CHAR, r, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << message << std::endl;
      }
  }
```

cf 2 slides before

Single Program, Multiple Data (SPMD)!

DEMO

A Sorted "Hello, World!" in Pictures




```
$ mpiexec -np 4 ./mpihello
Hello from 0 of 4
Hello from 1 of 4
Hello from 2 of 4
Hello from 3 of 4
```

MPI_Send

How is it used?

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

- Usage example

```
MPI_Send (&message[0], max_string , MPI_CHAR,
          0, 0, MPI_COMM_WORLD);
```

!

The first three arguments, **buf**, **count**, and **datatype**, determine the contents of the message. The remaining arguments, **dest**, **tag**, and **comm**, determine the destination of the message.

MPI_Recv

How is it used?

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

- Usage example

```
MPI_Recv (&message[0], max_string , MPI_CHAR,
          r, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
```

Point to Point Arguments

For MPI_Send & MPI_Rcv

- `buf` is the **array** storing the data to send or **ready to receive data**
- `count` states how many **replicas** of the **data type** will be sent, or the maximum allowed in when sending/receiving
- `source` and `dest` are **ranks** identifying the **target sender** or **receiver**
- `tag` is used to **distinguish messages** traveling on the **same connection** (we won't use)

Data Types

- MPI needs to know what kind of message it is delivering
- Since C/C++ types (int, char, and so on) can't be passed as arguments to functions, MPI defines a special type, `MPI_Datatype`, that is used for the datatype argument
- MPI also defines a number of constant values for this type

MPI_CHAR	MPI_UNSIGNED_CHAR	MPI_FLOAT
MPI_SHORT	MPI_UNSIGNED_SHORT	MPI_DOUBLE
MPI_INT	MPI_UNSIGNED	MPI_LONG_DOUBLE
MPI_LONG	MPI_UNSIGNED_LONG	MPI_BYTE

~~`MPI_string` does not exist!!~~

```
> mpiexec -np 11 -oversubscribe hello2
```

H	e	l	l	o		f	r	o	m		1	0		o	f		1	1	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17		

`tag`

- `tag` is a nonnegative `int`. It can be used to **distinguish messages that are otherwise identical**
- For example, suppose process 1 is sending `float`s to process 0:
  - some of the `float`s should be printed, while others should be used in a computation
  - but the first four arguments to `MPI_Send` provide no information regarding which `float`s should be printed and which should be used in a computation
    - process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation

!

`status`

- Detailed information on received data (low level details, see MPI specification)

- In many cases (for us!) it won't be used by the calling function and, as in our "hello" program, the special MPI constant `MPI_STATUS_IGNORE` can be passed

in many cases

Message Matching

So that a message sent by q can be received by r.

- In process *q* :

```
MPI_Send(send_buf, send_count, send_datatype, dest,
send_tag, send_comm);
```

- In process *r* :

```
MPI_Recv(recv_buf, recv_count, recv_datatype, src,
recv_tag, recv_comm, &status);
```

- The message sent by *q can be* received by *r* if:

  `send_comm = recv_comm, dest = r, src = q` and
  `recv_tag = send_tag`

- These conditions aren't quite enough:
  - if `recv_datatype = send_datatype` and
    `recv_count >= send_count`,
    then the message sent by *q* can be successfully received by *r*

Non-overtaking messages

- If process *q* sends two messages to process *r*, then the first message sent by *q* must be available to *r* before the second message

- There is **no restriction on the arrival of messages sent from different processes**:
  - if *q* and *t* both send messages to *r*, then even if *q* sends its message before *t* sends its message, there is no guarantees that *q*'s message become available to *r* before *t*'s message

Deadlocks in MPI

- **Deadlocks** occur when processes block for communication, but their requests remain unmatched or otherwise unprocessed

- Example:

Process 0	Process *n*
`MPI_Send (n)`	`MPI_Send (0)`
`MPI_Recv (n)`	`MPI_Recv (0)`

- Two approaches to prevent deadlocks:
  - either you smartly rearrange communication
  - use non-blocking calls (advanced topic, you will see in APSC)

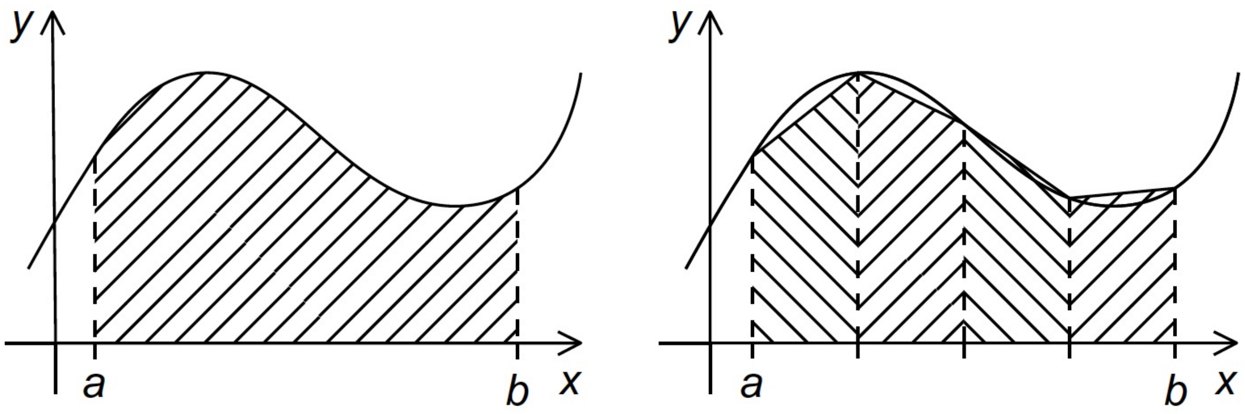
What is mainly suggested to do in APC.

Process Hang

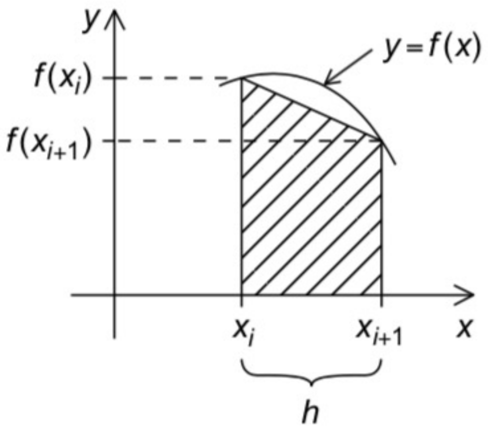
- If a process tries to receive a message and there's **no matching send**, then the process will **block forever**
- When you design your programs, be sure that every receive has a matching send

- Be very careful that there are no inadvertent mistakes in calls to `MPI_Send` and `MPI_Recv`
  - If the tags don't match, or **if the rank of the destination process is the same as the rank of the source process**, the receive won't match the send
  - Either a process will hang, or the receive may match *another* send!

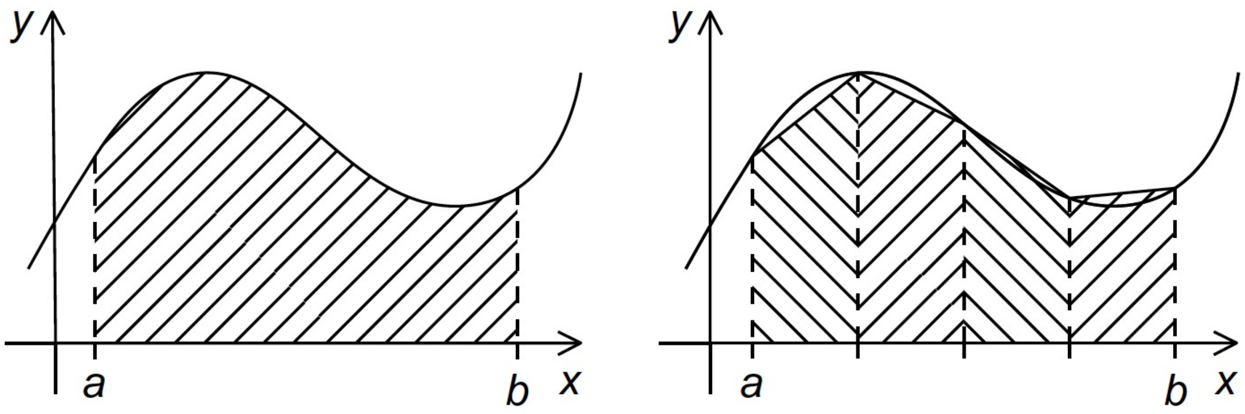
The Trapezoidal Rule in MPI



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



The Trapezoidal Rule in MPI



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

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

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

The Trapezoidal Rule in MPI

```
// quadrature function
// input: a, b, n

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

The Trapezoidal Rule in MPI

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

we want to use P.C

  - use many trapezoids, and we will use many more trapezoids than cores
  - at the end, we need to aggregate the computation of the areas of the trapezoids
- Basic idea:
  - split the interval [*a*, *b*] up into `comm_sz` subintervals
  - if `comm_sz` evenly divides *n* the number of trapezoids (and we will rely on this assumption initially), we can simply apply the trapezoidal rule with *n* / `comm_sz` trapezoids to each of the `comm_sz` subintervals
  - at the end, one processes, say 0, add the estimates

The Trapezoidal Rule in MPI - Pseudocode

```
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_int = quadrature(local_a, local_
if (my_rank != 0)
        Send local_int to process 0;
else{// my rank == 0
        total = local_int;
        for (proc = 1; proc < comm_sz; proc++) {
                Receive local integral from proc;
                total += local_int;
        }
}
if (my rank == 0)
        print result;
```

send to 0.

Aggregate

Variables whose contents are significant to all the processes are sometimes called **global** variables

Local variables are variables whose contents are significant only on the process that's using them

DEMO

MPI output

HOW TO AVOID RANDOMNESS ?

- In "Hello World" and the trapezoidal rule programs, process 0 writes to the standard output
- 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 standard output and error
  - but output is random !
- To have "sorted output", the common practice is each process sends its output to process 0, and process 0 can print the output in process rank order

MPI Input

- Unlike output, most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to standard input

- The common practice is that process 0 performs `std::cin` and then it broadcasts, or scatters, input values to all processes

- It’s time to consider **collective communication** then!