deep se
dependable evolvable pervasive software engineering group

Message Passing Interface

Eugenio Gianniti & Danilo Ardagna

Politecnico di Milano
name.lastname@polimi.it

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

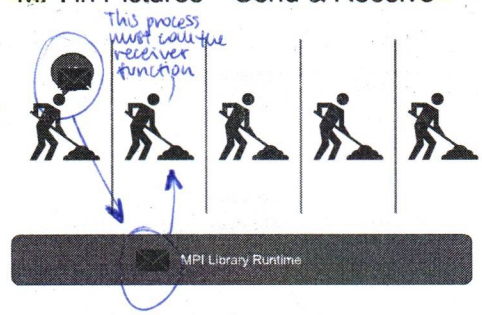
- In MPI all parallelism is explicit
  - Identifying what should be parallelized and how is on programmers
- MPI was designed for distributed memory architectures, even if the implementations currently support any common parallel architecture
- Hence, MPI virtually allows running your code on any parallel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

Message Passing Basic Idea

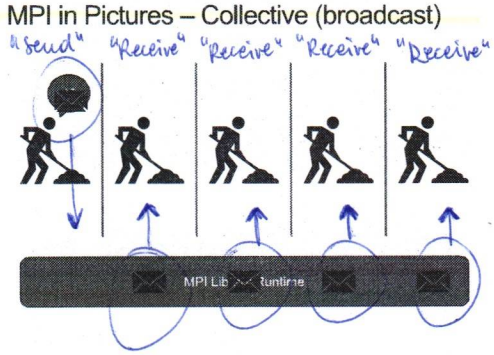
- In message-passing programs, a program running on one core is usually called a process
- Two processes can communicate by calling functions:
  - one process calls a *send* function
  - the other calls a *receive* function
- MPI supports also global communication functions that can involve more than two processes
  - These functions are called collective communications

MPI in Pictures – Send & Receive

If this process wants to send a message, it calls the send function, create the message (with the address of the receiver)



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 *`

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;
}
```

These are standard names: we cannot pass other names! we cannot say (e.g.) int main(int a,..)

If we run the program without any arguments then argc = 1 and argv[0] is the pointer to the name of the program. All the other parameters are passed starting from argv[1]

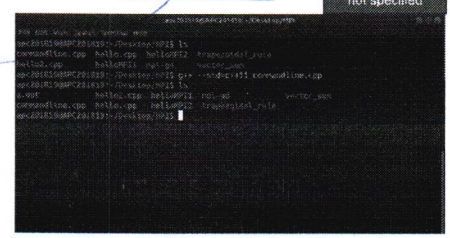
Compiling your C++ program

a. `$ g++ --std=c++11 file_name [-o executable]`

b. `$ g++ --std=c++11 commandline.cpp`

Default is a.out if not specified

name_of_the_file.cpp



1. We write the code in atom (comparable to Clion / Dev)
2. We go to commandline and we compile:
   - (a.) we're adding the name to the executable
   - (b.) the executable has a standard name: a.out
3. We run the code: (the executable)
   - (b.) ./a.out arguments
   - (a.) ./executable arguments

Executing your program at the command line

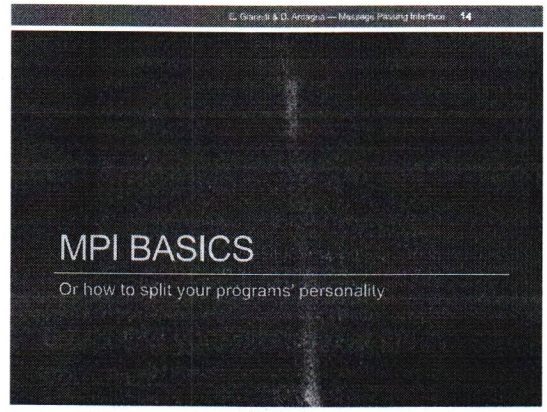
argc = 2
argv[1] = "testing"

```
$ ./a.out testing
The argument supplied is testing

$ ./a.out testing1 testing2
Too many arguments supplied

$ ./a.out
One argument expected
```

- 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



Hello World! (C language)

```
#include <cstdio>

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

let's say hello from each processor!

Hello World!

that's because we do C (and not C++)

```
#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 ();
}
```

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

from here we'll know how many processes are running (size (after the call) will contain it)

! MPI_Init provides arguments across all the processes
The arguments are shared among different processes thanks to MPI_Init

(after the call) rank will be the process ID

! whenever we don't need to communicate anymore

Compile & Run

name of the executable

- To compile MPI code you use:

```
$ mpicxx -o exe file1.cc [file2.cc ...]
```

  - All the flags are the same as with `g++`

command to compile

list of the names of the files that we have to execute (compile)
(multiple .cpp if we have a code which is spread in multiple source files (e.g. if we have more classes))

- To run MPI executables you use:

```
$ 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`

command to run in parallel

number of processes = 4

name of the executable

Note: if we have more processes than the physical cores, the processes will alternate in the execution on the physical resources

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:

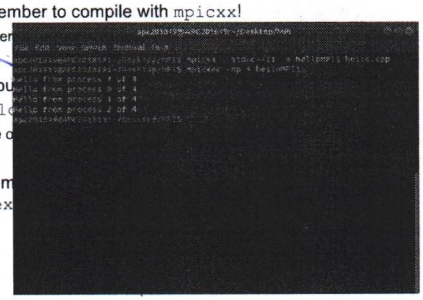
```
Hello from process 0 of 4
```

  - 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`!
  - Other
- The ou
  Hell
  - The o
- How m
  mpiex



first line:
$ mpicxx -o hello_exe --std=c++11 hello.cpp
we're compiling the file hello.cpp in C++ and we're calling the executable "hello_exe"

second line:
$ mpiexec -np 4 hello_exe
we're running on 4 parallel processes the executable "hello_exe"
what we get:
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 3 of 4
Hello from process 2 of 4
The order is random!

MPI_Init and MPI_Finalize

- `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`

It sets up the communication channels

MPI_Init and MPI_Finalize

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

(pointer to integer, pointer to char)

- 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

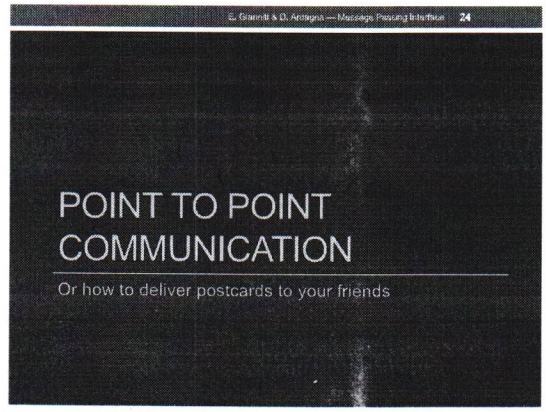
- `MPI_Finalize` tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed

```
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;
}
```

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

point to point communications are the ones that involve only send and receives

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)

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

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

- 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

* It's the name of our universe of processes (it's the name of the set of processes from which we use processes)

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;
      }
  }
```

DEMO

Every process has access to the standard output of the program → we need to delegate one single process to print out (in this way we get control of the output)

Note: until we run the process we don't know how many processes we'll use; we're sure that we'll run at least one process → we delegate rank 0 to write the output (print out). Every process will send the output to rank 0, then rank 0 needs to receive the output.

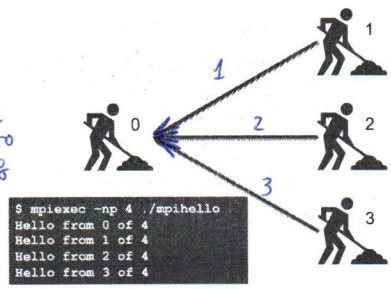
If the process is not rank 0 then it must send the message: which one? `&message[0]` (we provide the address of the 1st character)
how long? at most max_string
what type? MPI_CHAR (characters)
to who? receiver ranked "0"
through what? MPI_COMM_WORLD *
(there is an additional param that we set to 0)

If the process is rank 0 then it builds its own message (`std::cout << message << std::endl;`). Then it enters a receiver loop and it receives all the messages. It stores the messages in "message" by providing the first character. **Note**: it stores one message at the time.

rank of the receiver

this means that we're not interested in the errors that we can get from this call

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
```

rank 0 prints its own message and then waits to receive something from rank 1, then rank 2 and then rank 3

MPI_Send

```
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.

pointer to void = pointer to anything (in C language)
how many elements
type of the elements
↳ **what** we are sending

who is receiving the message
process destination
communicator

MPI_Recv

```
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 from which we start storing the data we're receiving
how many data we're receiving
type of the receiving data
dual interface (to MPI_Send)

sender communicator
status (data structure with the error codes)

Point to Point Arguments

- `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`

These are from C (~~C++~~)
(we don't have strings, vectors, ..)

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

Message Matching

- 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

- 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)`

both will remain blocked until the other receives what they're sending → blocked forever

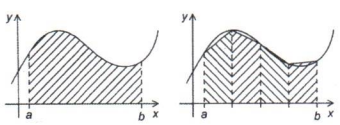
- Two approaches to prevent deadlocks:
  - either you smartly rearrange communication
  - use non-blocking calls (advanced topic, you will see in APSC)

→ they introduce RACE CONDITION

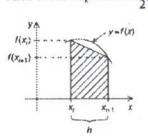
Process Hang

- 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!

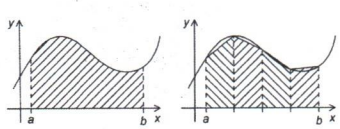
The Trapezoidal Rule in MPI



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



The Trapezoidal Rule in MPI



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

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

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

Sequential version:

```
// 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
  - 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

We divide the whole interval in subintervals. How many? As the number of cores. Then, each subinterval will be treated by one process (split, etc. ...)

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_b, local_n, h);
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;
```

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

= a + my_rank * local_n * h;

= quadrature(local_a, local_b, local_n, h);

! We call **LOCAL** the variables which have a specific value according to the process we're considering. (e.g. here local_a, local_b)

Every process evaluate the integral on its subset and then sends to process rank 0. Process rank 0 evaluates the sum

DEMO

MPI output

- 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!