Introduction to Message-Passing Interface

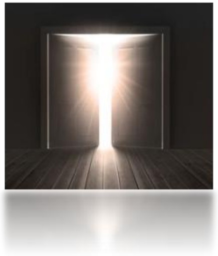
Luca Tornatore - I.N.A.F.

DATA SCIENCE & ARTIFICIAL INTELLIGENCE

SCIENTIFIC & DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

Outline


Intro to MPI

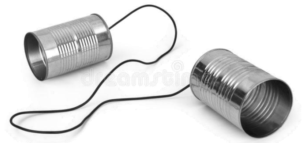
Point-to-Point Communications

Collective Communications

few JEDI things

Outline

Intro to MPI


Point-to-Point Communications

Collective Communications

few JEDI things

Why MPI

As we have seen in the “Introduction to parallelism”, the Message-Passing Interface implements a standard that refers to the distributed-memory paradigm of parallel programming.

Hence: MPI implements the *distributed memory* parasigm, and deals with how to *move data* among separate address spaces.

Why MPI

- The MPI *tasks* are separate processes, each having its own address space
  - the programmer manages the memory as it was a serial code

- Tasks communicate for
  - Synchronization
  - Data movement

- The MPI tasks exchange data *only* by collaborative *explicit messages* (*)
  - the programmer is in charge of making available memory regions
  - every data distribution among tasks also happens via explicit messages, also managed by the programmer

(*) Actually, since MPI-2, but at mature level from MPI-3, direct memory access is possible among tasks. These are *called one*-sided communications exactly because they do not require the co-operation of the tasks. We'll se some details in week2

What is MPI

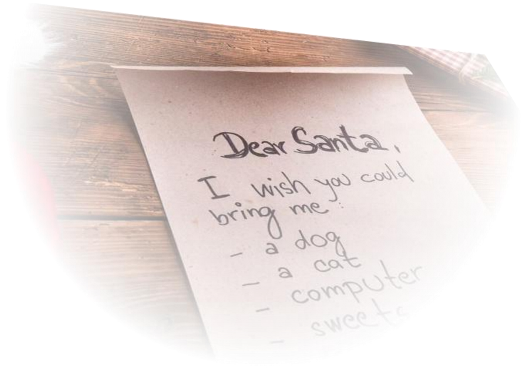
Let’s specify that MPI

- It is “a book”, i.e. the specification of a standard, *not* an implementation of it.
- Defines **a library, not a language**. All operations depends on the execution of routines.
- It defines the API and the behaviour of the functions; your FORTRAN / C / C++ program is compiled by your compiler after a wrapping by the mpicc program, and then linked to that library.

Let’s invent MPI

If you have to “send a message”,
let’s say a mail..

What do you need ?

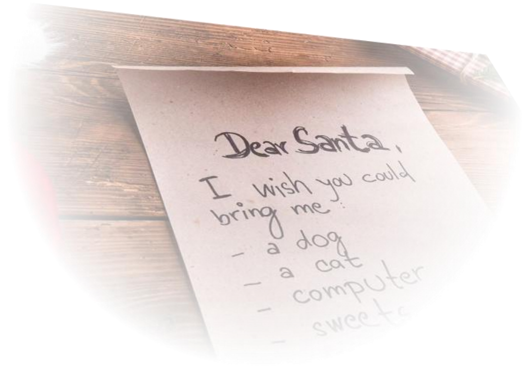


Dear Santa,
I wish you could bring me :
- a dog
- a cat
- computer
- sweet

Let's invent MPI

If you have to "send a message", let's say a mail..
What do you need ?

1. a paper **sheet**
2. an **envelop**
3. the name of the **receiver**
4. the **sender**'s name, to get an answer (or a receipt acknowledgment)
5. and, of course a **postal service**
   a) postal office
   b) postal network
   c) postpersons



Let’s invent MPI

Well, that is, in general, exactly how MPI works..

```
char  *data_region = (char*)malloc( ... );

// do something on the data region

SEND_A_MESSAGE( data_region, how_may_sheets, to_who, in_what_country );
```

the **paper sheet**

how long

the receiver
his “name” is also the address if there are only unique names

*The community, or “framework” we are referring to.
You may belong to different communities, right?*

Let’s invent MPI

When you receive a letter, however, in order to get it at the right place - on your desk, in your files, … - you have to perform some actions.
It may be that you just get a notifcation, and you have to walk to the postal office, or the you simply have to check your inbox and pick up what is inside.

In short, to receive a letter, you must be collaborative and perform some tasks:

```
RECV_A_MESSAGE( where_to_put_it,
                how_long,
                from_who,
                in_what_country );
```

where_to_put_it ← of course you need some room for it

how_long ← and you need to know how long it is

from_who ← better to know who is the sender

in_what_country ← and the “framework” he belongs to

Let's invent MPI

So, we expect that there must be **point-to-point**

- a way to **send** messages
- a corresponding way to **receive** messages
- a way to know "the **names**" in "the **framework**"

what if the receiver *does not* expect a message?

Let’s invent MPI

Since sometimes we need to broadcast messages, it would be nice if

- there was a **broadcasting** (**one-to-many**)

« doomsday will be at this time at this place rsvp »

Since we may also in the need of receiving answers, it would also be nice if

- there was a **collection** mechanism (**many-to-one**) mechanism
- the answer/collection was possible **many-to-many** (try to organize a meeting without that..)

« I won’t be able to attend sincerly»

Let's invent MPI

We're lucky, because MPI designers thought the same.

And that is exactly what we are covering in the next days: P2P communications and collective communications.

They exist in several different flavours, of which we'll explore definition and usage.

The very basic

However, we need to start with MPI before using its routines.
And that is the most simle MPI code you may ever write:

```
#include <mpi.h>
int main ( int argc, char **argv )
{
  MPI_Init( &argc, &argv );

  MPI_Finalize();
  return 0;
}
```

just try it:

```
mpicc verybasic.c -o verybasic
mpirun -np $NUM ./verybasic
```

The very basic

raise your hand if you have any doubts about these two things..

```
#include <mpih>

int main ( int argc, char **argv )
{
  MPI_Init( &argc, &argv );

  MPI_Finalize();
  return 0;
}
```

All MPI routine calls begin with **MPI_**

You always need to **initialize MPI** first.
Until we don't do that, you'll just have a bunch of processes doing the same thing, and uch probably failing at the first MPI call.

Best practice: always finalize your environment

Best practice: always explicitly return, and return with a value

The very basic

Better to get used to **MPI_Init_thread()**

```
int main ( int argc, char **argv )
{
  int mpi_provided_thread_level;
  MPI_Init_thread( &argc, &argv, REQUIRED_LEVEL, &mpi_provided_thread_level );

  if ( mpi_provided_thread_level < REQUIRED_LEVEL ) {
      ..manage the situation.. }
...
```

where **REQUIRED_LEVEL** can be:

```
MPI_THREAD_SINGLE       Only one thread will execute.
MPI_THREAD_FUNNELED     Only the thread that called MPI_Init_thread will make MPI calls.
MPI_THREAD_SERIALIZED   Only one thread will MPI library calls at one time.
MPI_THREAD_MULTIPLE     Multiple threads may call MPI at once with no restrictions.
```

The very basic: init and end

The second most simle MPI code you may ever write, is, obviosouly “hello MPI world”

```
#include <mpi.h>

int main ( int argc, char **argv )
{
  int provided_thread_level;
  MPI_Init_thread( &argc, &argv, MPI_THREAD_SINGLE, &provided_thread_level );

  printf(“hello MPI world\n”);

  MPI_Finalize();
  return 0;
}
```

Let’s try that and see what happens

The very basic: who am I?

Definitely something was missing, and the following is the real "hello MPI world" thing

```
int main ( int argc, char **argv )
{
  int Ntasks, Myrank;
  int provided_thread_level;
  MPI_Init_thread( &argc, &argv, MPI_THREAD_SINGLE, &provided_thread_level );

  MPI_Comm_size ( MPI_COMM_WORLD, &Ntasks );
  MPI_Comm_rank ( MPI_COMM_WORLD, &Myrank );
  printf("hello there MPI world from task %d out of %d\n",
          Myrank, Ntasks );

  MPI_Finalize();
  return 0;
}
```

see `helloworld.c`

The very basic: who am I ?

Definitely something was missing, and the following is the real “hello MPI world” thing

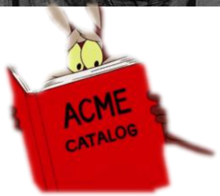
```
int main ( int argc, char **argv )
{
  int Ntasks, Myrank;
  int provided_thread_level;
  MPI_Init_thread( &argc, &argv, MPI_THREAD_SINGLE, &provided_thread_level );

  MPI_Comm_size ( MPI_COMM_WORLD, &Ntasks );
  MPI_Comm_rank ( MPI_COMM_WORLD, &Myrank );
  printf(“hello there MPI world from task %d out of %d\n”,
          Myrank, Ntasks );

  MPI_Finalize();
  return 0;
}
```

Gets how many tasks are in the communicator

Gets the rank of the calling process in the communicator

The very basic: an hint

That is the moment for an **hint**..



There exist a wonderful thing, invented by smart people for people like me, which can not remember too complicate stuff..

the **man page**!

Try `man MPI_$the_routine_you_want_to_discover`

The very basic: the communicators

Back to here… what is "a communicator" ?

```
int main ( int argc, char **argv )
{
  int Ntasks, Myrank;
  MPI_Init_thread( &argc, &argv, ... );

  MPI_Comm_size ( &Ntasks, MPI_COMM_WORLD );
  MPI_Comm_rank ( &Myrank, MPI_COMM_WORLD );
  printf("hello there MPI world from task %d out of %d\n",
          Myrank, Ntasks );

  MPI_Finalize();
```

Gets how many tasks are in the **communicator**

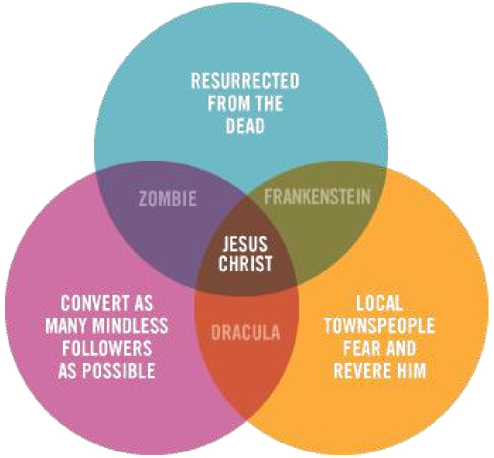
The very basic: the communicators

Communicators and **groups** are a very central concepts in MPI.

Tasks can form **groups**:

A task can be in more than one **group** at a time

The same group can be in different situations


RESURRECTED FROM THE DEAD
ZOMBIE
FRANKENSTEIN
JESUS CHRIST
CONVERT AS MANY MINDLESS FOLLOWERS AS POSSIBLE
DRACULA
LOCAL TOWNSPEOPLE FEAR AND REVERE HIM

You and your colleagues at work and at a rave

(didn't put a picture for privacy reasons)

picture from https://clipart-library.com/clipart/grouping-cliparts-7.htm

The very basic: the communicators

Communicators and **groups** are a very central concepts in MPI.

The same group can be in different situations

You and your colleagues at work and at a rave

(didn't put a picture for privacy reasons)

A **COMMUNICATOR** is the combination of a group and its "context".

You can build as many groups as you want, and they may or not have a communicator.

However, if you want to communicate among tasks in a group, you need a communicator,

(for instance, to assign a rank to every task in the group; for instance your name may be "Gandalf" at work, but to address you at the party we need to call you "Nuanda")

The very basic: the communicators

Communicators and **groups** are a very central concepts in MPI.

A **COMMUNICATOR** is the combination of a group and its “context”.

- This functionality offers the capability of isolating communication between application modules with an effective “sandbox” for different contexts.
  For instance, a parallel library and your application will use internally their own communicator, separating contexts.
- By creating groups of MPI processes, that may or not overlap with each other, it is possible to
  - separate contexts within different modules of the same application (useful or even *advisable*)
  - express multiple levels of parallelism

The very basic: the communicators

Communicators and **groups** are a very central concepts in MPI.

A **COMMUNICATOR** is the combination of a group and its “context”.

- `MPI_COMM_WORLD`
  is the default communicator available right after the call to MPI_Init.
  Its group contains *all* the tasks started by your job.
- `MPI_COMM_NULL`
  signals an invalid / non existent communicator
- `MPI_COMM_SELF`
  contains only the process itself
- `MPI_GROUP_NULL`
  signals an invalid / non existent group

The very basic: the communicators

BEST PRACTICE
always create a separated “context” for the application you’re writing

```
#include <mpi.h>

int main ( int argc, char **argv )
{
   int Myrank, Ntasks;
   int mpi_provided_thread_level;
   MPI_Comm myCOMM_WORLD;

   MPI_Init_thread ( &argc, &argv, MPI_THREAD_SINGLE, &mpi_provided_thread_level);
   MPI_Comm_dup ( MPI_COMM_WORLD, &myCOMM_WORLD );

   MPI_Comm_size( &Ntasks, myCOMM_WORLD);
   MPI_Comm_rank( &Myrank, myCOMM_WORLD);
   ...
```

The very basic: the communicators

BEST PRACTICE
always create a separated "context" for the application you're writing

Quoting Victor Eijkhout:

« Imagine you're writing a library, and your library makes MPI calls. Now imagine that some Isend and Irecv calls are done in a library routine that the user calls, with the Wait calls in another routine that the library calls.

Since user code is active in between the library doing Isend/Irecv and the library doing wait, it is possible for the user to catch the library Isend/Irecv calls, or conversely for the user to start Isend/Irecv calls and the library to catch them.

You prevent such mishaps by letting the library use a duplicate communicator on the same group of processes. The communicator in effect becomes a label on the messages. »