High-Performance Mathematics

Programming in a distributed setting: MPI

Progetto Speciale per la Didattica 2023/24

Fabio Durastante (L2)

April 17, 2024

Dipartimento
di Matematica
Università di Pisa

Table of Contents

1 Distributed parallelism in practice

▶ Distributed parallelism in practice

▶ An Introduction to MPI
Preliminary work
Our First MPI Program
The MPI parallel environment
When to travel the MPI route

How do we realize practically this parallelism?

Let us focus on what we have discussed until now:

- We have "**machines**" with multiple processors and whose main memory is partitioned into fragmented components,
- We have **algorithms** that can divide a problem of size N among these processors so that they can run (almost) independently,
- With a certain degree of approximation, we know how to compute what is the *best improvement* we can expect from a parallel program with M processors on a problem of size N:

  Strong scaling: fixed problem size, increasing number of processes, Amdahl's law;
  Weak scaling: fixed problem size per computing process, Gustafson's law.

How do we realize practically this parallelism?

What we need to discuss now is then:

> "How can we actually implement these algorithms on *real machines*?"

How do we realize practically this parallelism?

What we need to discuss now is then:

> "How can we actually implement these algorithms on *real machines*?"

- We need a way to define a **parallel environment** in which every processor is accounted for,

How do we realize practically this parallelism?

What we need to discuss now is then:

> "How can we actually implement these algorithms on *real machines*?"

- We need a way to define a **parallel environment** in which every processor is accounted for,
- We need to have **data formats** that are aware of the fact that we have a *distributed* memory,

How do we realize practically this parallelism?

What we need to discuss now is then:

> "How can we actually implement these algorithms on *real machines*?"

- We need a way to define a **parallel environment** in which every processor is accounted for,
- We need to have **data formats** that are aware of the fact that we have a *distributed* memory,
- We need to **exchange data** between the various memory fragments.

Table of Contents

2 An Introduction to MPI

▶ Distributed parallelism in practice

▶ An Introduction to MPI
- Preliminary work
- Our First MPI Program
- The MPI parallel environment
- When to travel the MPI route

Message Passing Interface – www.mpi-forum.org

"MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists." – W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22 (6), 1996.

Message Passing Interface – www.mpi-forum.org

"MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists." – W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22 (6), 1996.

- MPI implementations consist of a specific set of routines directly callable from C, C++, Fortran;

Message Passing Interface – www.mpi-forum.org

"MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists." – W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22 (6), 1996.

- MPI implementations consist of a specific set of routines directly callable from C, C++, Fortran;
- MPI uses *Language Independent Specifications* for calls and language bindings;

Message Passing Interface – www.mpi-forum.org

"MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists." – W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22 (6), 1996.

- MPI implementations consist of a specific set of routines directly callable from C, C++, Fortran;
- MPI uses *Language Independent Specifications* for calls and language bindings;
- The MPI interface provides an essential *virtual topology*, synchronization, and communication functionality inside a set of processes.

Message Passing Interface – www.mpi-forum.org

"MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists." – W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22 (6), 1996.

- MPI implementations consist of a specific set of routines directly callable from C, C++, Fortran;
- MPI uses *Language Independent Specifications* for calls and language bindings;
- The MPI interface provides an essential *virtual topology*, synchronization, and communication functionality inside a set of processes.
- There exist **many implementations** of the MPI specification, e.g., MPICH, Open MPI, pyMPI, Spectrum MPI, Intel MPI, . . .

Fallacies of distributed computing

2 The network is reliable;
1 Latency is zero;
5 Bandwidth is infinite;
4 The network is secure;
3 Topology doesn't change;
6 There is one administrator;
8 Transport cost is zero;
7 The network is homogeneous.

Peter Deutsch

All prove to be **false** in the long run and all cause **big trouble** and **painful** learning **experiences**.

Preliminary work

Let's start with **a preliminary setup** and connect to a machine that is capable of producing the executables we need:

```
ssh n.cognomeXX@a3-dottY.cs.dm.unipi.it
```

where

- `n.cognomeXX` are your "*credenziali di ateneo*",
- and `Y` $= 1, 2, \ldots$ is one of the machines of *Aula DM3*.

Already in *Aula DM3*.

If you are already physically connected to one of the machines of *Aula DM3*, you can skip this passage and just open a terminal.

Putting up a `git` repository for our code

To **develop our code** and track our progress, we set up a `git` repository with the results.

Putting up a `git` repository for our code

To **develop our code** and track our progress, we set up a `git` repository with the results.

1. Go to: git.phc.dm.unipi.it,

Putting up a `git` repository for our code

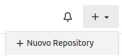
To **develop our code** and track our progress, we set up a `git` repository with the results.

1. Go to: git.phc.dm.unipi.it,
2. Login to the system: Accedi

Putting up a `git` repository for our code

To **develop our code** and track our progress, we set up a `git` repository with the results.

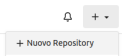
1. Go to: git.phc.dm.unipi.it,
2. Login to the system: 

3. Create a **new repository**: 

Putting up a `git` repository for our code

To **develop our code** and track our progress, we set up a `git` repository with the results.

1. Go to: git.phc.dm.unipi.it,
2. Login to the system:

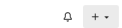
3. Create a **new repository**: 

We must now **select the settings** necessary to define the repository:

Putting up a `git` repository for our code

To **develop our code** and track our progress, we set up a `git` repository with the results.

1. Go to: git.phc.dm.unipi.it,
2. Login to the system:

3. Create a **new repository**: 

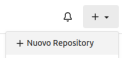
We must now **select the settings** necessary to define the repository:

- The *unique* (for our account) repository name:

Putting up a `git` repository for our code

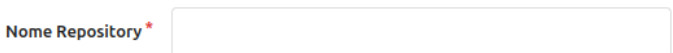
To **develop our code** and track our progress, we set up a `git` repository with the results.

1. Go to: git.phc.dm.unipi.it,
2. Login to the system: 

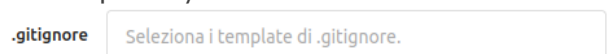
3. Create a **new repository**: 

We must now **select the settings** necessary to define the repository:

- The *unique* (for our account) repository name:

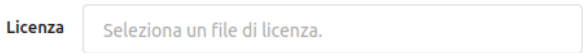
  

- A `.gitignore` template, that will simplify the selection of file we wish to preserve on the repository. We can select `c` code:

Putting up a `git` repository for our code

- We now need to select a **license for our code**:



A **good starting point** to decide what license we may need is visiting the website: choosealicense.com another set of useful information is available on Wikipedia.

GPL License

MIT License

BSD License

Putting up a `git` repository for our code

- We now need to select a **license for our code**:

Licenza | Seleziona un file di licenza.

A **good starting point** to decide what license we may need is visiting the website: choosealicense.com another set of useful information is available on Wikipedia.

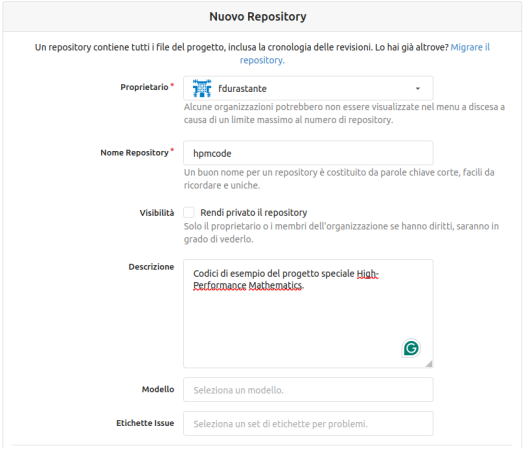
GPL License

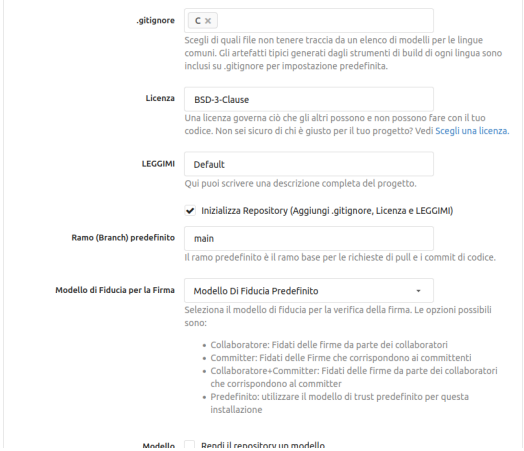
MIT License

BSD License

Be sure that this option is on: ☑ Inizializza Repository (Aggiungi .gitignore, Licenza e LEGGIMI) .

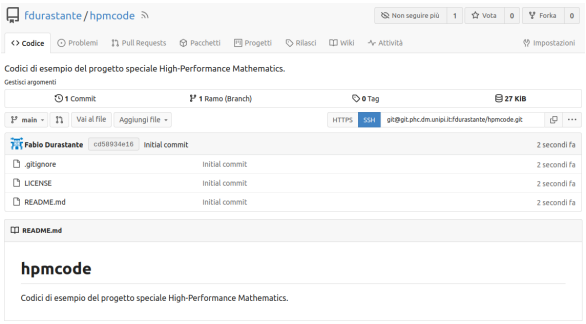
Putting up a `git` repository for our code





And then push: Crea Repository

Putting up a `git` repository for our code



Let's clone the repository we created on the machine:

```
cd Documents
git clone git@git.phc.dm.unipi.it:fdurastante/hpmcode.git
cd hpmcode
```

the link should be the one of your repository, not mine!

Hello (parallel) world!

In today's lecture we are going to use the MPI inside C programs, and start writing:

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

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

- `#include "mpi.h"` provides basic MPI definitions and types,
- `MPI_Init` start MPI, it has to precede any MPI call!
- `MPI_Finalize` exits MPI
- All the non–MPI routines are local!

We need to **save the code into the Git repository folder**.

Hello (parallel) world! – Compile, Link and Run

We need now to *compile* and *link* the `helloworld.c` program.

- We need to **set-up the environment** that will contain a **compiler** and an **implementation of MPI**.

Hello (parallel) world! – Compile, Link and Run

We need now to *compile* and *link* the `helloworld.c` program.

- We need to **set-up the environment** that will contain a **compiler** and an **implementation of MPI**.

</> To this end, we use environment module.

Hello (parallel) world! – Compile, Link and Run

We need now to *compile* and *link* the `helloworld.c` program.

- We need to **set-up the environment** that will contain a **compiler** and an **implementation of MPI**.

</> To this end, we use environment module.

Environment Module

The Modules package is a tool that simplifies shell initialization and lets users easily modify their environment during a session using *modulefiles*.

Modules can be **loaded** and **unloaded** dynamically and atomically, in an clean fashion.

Modules are useful in managing different versions of applications. Modules can also be bundled into meta-modules that will load an entire suite of different applications.

Hello (parallel) world! – Compile, Link and Run

To discover what module we have available, we can run the command:

```
module avail
```

That will answer us:

```
--------------- /software/spack/share/spack/modules/linux-ubuntu22.04-zen3 ---------------
amdblis/4.2-aocc-4.2.0                    hpctoolkit/2023.08.1-openmpi-5.0.2-gcc-11.4.0
amdfftw/4.2-openmpi-5.0.2-aocc-4.2.0      libflame/5.2.0-aocc-4.2.0
amdlibm/4.2-aocc-4.2.0                    openmpi/5.0.2-cuda-11.8.0-aocc-4.2.0
amdscalapack/4.2-openmpi-5.0.2-aocc-4.2.0 openmpi/5.0.2-cuda-12.3.0-gcc-11.4.0
amduprof/4.2.850-aocc-4.2.0               openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0
aocc/4.2.0                                petsc/3.20.4-openmpi-5.0.2-gcc-12.2.0
aocl-sparse/4.2-aocc-4.2.0                py-torch/2.2.1-openmpi-5.0.2-gcc-11.4.0
cuda/11.8.0-aocc-4.2.0                    suite-sparse/7.3.1-cuda-12.3.0-gcc-11.4.0
cuda/12.3.0-gcc-11.4.0                    suite-sparse/7.3.1-cuda-12.3.0-gcc-12.2.0
cuda/12.3.0-gcc-12.2.0                    vtk/9.2.6-openmpi-5.0.2-gcc-12.2.0
gcc/12.2.0
```

From which we discover that we have **different available compilers**.

Hello (parallel) world! – Compile, Link and Run

Let us **load** the `gcc/12.2.0` compiler together with the `openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0` implementation of MPI:

```
module load gcc/12.2.0 openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0
```

this will make us available the compiler to produce MPI executable:

```
mpicc helloworld.c -o helloworld
```

- `mpicc` is a **wrapper** for a C compiler provided by the implementation of MPI we are using.
- the option `-o` sets the name of the compiled (executable) file.

Hello (parallel) world! – Compile, Link and Run

Let us **load** the `gcc/12.2.0` compiler together with the `openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0` implementation of MPI:

```
module load gcc/12.2.0 openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0
```

this will make us available the compiler to produce MPI executable:

```
mpicc helloworld.c -o helloworld
```

Let us see what is happening behind the curtains

- you can first try to discover what compiler are you using by executing `mpicc --version`, that will give you:

```
gcc (Spack GCC) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Hello (parallel) world! – Compile, Link and Run

Let us **load** the `gcc/12.2.0` compiler together with the `openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0` implementation of MPI:

```
module load gcc/12.2.0 openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0
```

this will make us available the compiler to produce MPI executable:

```
mpicc helloworld.c -o helloworld
```

Let us see what is happening behind the curtains

- you can first try to discover what compiler are you using by executing `mpicc --version`,
- or discover what are the library inclusion and linking options by asking for `mpicc --showme:compile` and `mpicc --showme:link`, respectively.

Hello (parallel) world! – Compile, Link and Run

Let us **load** the `gcc/12.2.0` compiler together with the `openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0` implementation of MPI:

```
module load gcc/12.2.0 openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0
```

this will make us available the compiler to produce MPI executable:

```
mpicc helloworld.c -o helloworld
```

Let us see what is happening behind the curtains

- you can first try to discover what compiler are you using by executing `mpicc --version`,
- or discover what are the library inclusion and linking options by asking for `mpicc --showme:compile` and `mpicc --showme:link`, respectively.
- In general, looking at the output of the `man mpicc` command is always a good idea.

Hello (parallel) world! – Compile, Link and Run

A **piece of advice**: if your program is anything more realistic than a classroom exercise use `make`, and save yourself from writing painfully long compiling commands, and dealing with complex dependencies more than once.

> *"Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files and how to compute it from other files."*

A simple `Makefile` for our first test would be

```
MPICC = mpicc #The wrapper for the compiler
CFLAGS += -g  #Useful for debug symbols
all: helloworld
helloworld: helloworld.c
 $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
clean:
 rm -f helloworld
```

Hello (parallel) world! – Compile, Link and Run

If you are **running on your machine** (possibly for doing some *debug*), you can run your first parallel program by doing:

```
mpirun [ -np X ] [ --hostfile <filename> ]  helloworld
```

or by using its synonym

```
mpiexec [ -np X ] [ --hostfile <filename> ]  helloworld
```

- `mpirun`/`mpiexec` will run `X` copies of `helloworld` in your current run-time environment, scheduling (by default) in a round-robin fashion by CPU slot.
- if running under a supported resource manager, Open MPI's `mpirun` will usually automatically use the corresponding resource manager process starter, as opposed to, for example, rsh or ssh, which require the use of a hostfile, or will default to running all `X` copies on the localhost

Hello (parallel) world! – Compile, Link and Run

If you are **running on your machine** (possibly for doing some *debug*), you can run your first parallel program by doing:

```
mpirun [ -np X ] [ --hostfile <filename> ]  helloworld
```

or by using its synonym

```
mpiexec [ -np X ] [ --hostfile <filename> ]  helloworld
```

- `mpirun`/`mpiexec` will run `X` copies of `helloworld` in your current run-time environment, scheduling (by default) in a round-robin fashion by CPU slot.
- if running under a supported resource manager, Open MPI's `mpirun` will usually automatically use the corresponding resource manager process starter, as opposed to, for example, rsh or ssh, which require the use of a hostfile, or will default to running all `X` copies on the localhost
- as always, *look at the manual*, by doing `man mpirun`.

Hello (parallel) world! – Compile, Link and Run

If we now run

```
mpirun -np 6 helloworld
```

we get

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

Every process executes the line

```
printf("Hello, world!\n");
```

that it is a local routine!

Hello (parallel) world! – Compile, Link and Run

If we now run

```
mpirun -np 6 helloworld
```

we get

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

Every process executes the line

```
printf("Hello, world!\n");
```

that it is a local routine!

local versus non-local procedure

A procedure is **local** if completion of the procedure depends only on the local executing process.
A procedure is **non-local** if completion of the operation may require the execution of some MPI procedure on another process. Such an operation *may require communication* occurring with another user process.

Add, commit and push our working code to git

Now that we have a **working version of our first code**, it's time to **checkpoint it** on the git repository.

1. We first run `git status` obtaining:

```
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
(use "git add <file>..." to include in what will be committed)
   Makefile
   helloworld
   helloworld.c

nothing added to commit but untracked files present (use "git add" to track)
```

`Add`, `commit` and `push` our working code to `git`

Now that we have a **working version of our first code**, it's time to **checkpoint it** on the git repository.

1. We first run `git status` obtaining:
2. We discover that we can **add** to the repository the files `helloworld.c` and `Makefile`. We can do it with the command:

   ```
   git add helloworld.c Makefile
   ```

Add, `commit` and `push` our working code to `git`

Now that we have a **working version of our first code**, it's time to **checkpoint it** on the git repository.

1. We first run `git status` obtaining:
2. We discover that we can **add** to the repository the files `helloworld.c` and `Makefile`. We can do it with the command:

   ```
   git add helloworld.c Makefile
   ```
3. Then we can **commit** it to the *repository*

   ```
   git commit -m "My first MPI code"
   ```

Add, commit and push our working code to git

Now that we have a **working version of our first code**, it's time to **checkpoint it** on the git repository.

1. We first run `git status` obtaining:
2. We discover that we can **add** to the repository the files `helloworld.c` and `Makefile`. We can do it with the command:

   ```
   git add helloworld.c Makefile
   ```
3. Then we can **commit** it to the *repository*

   ```
   git commit -m "My first MPI code"
   ```
4. and **push** it to the repository:

   ```
   git push
   ```

Add, commit and push our working code to git

After it, we will get:

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 684 bytes | 342.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
remote: . Processing 1 references
remote: Processed 1 references in total
To git.phc.dm.unipi.it:fdurastante/hpmcode.git
cd58934..c049fb3  main -> main
```

Add, commit and push our working code to git

If we go looking to the website we see that the files are now stored there:

Fabio Durastante `c049fb3ec1` My first MPI code		2 minuti fa
.gitignore	Initial commit	48 minuti fa
LICENSE	Initial commit	48 minuti fa
Makefile	My first MPI code	2 minuti fa
README.md	Initial commit	48 minuti fa
helloworld.c	My first MPI code	2 minuti fa

Add, commit and push our working code to git

We can see what we have done with the repository with the command: `git log`.

```
commit c049fb3ec1865c367521e960259e7b47325ac02b (HEAD -> main, origin/main, origin/HEAD)
Author: Fabio Durastante <a037726@A3-dott7.polo2.sid.unipi.it>
Date:   Sun Apr 14 22:19:58 2024 +0200

My first MPI code

commit cd58934e167e6a141a1a7ce228b3a014b4badb15
Author: Fabio Durastante <fabio.durastante@unipi.it>
Date:   Sun Apr 14 19:34:17 2024 +0000

Initial commit
```

The MPI parallel environment

Let us modify our `helloworld` to investigate the MPI parallel environment. Specifically, we want to answer, from within the program, to the questions:

1. How many processes are there?
2. Who am I?

```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( "Hello world! I'm process %d of %d\n",rank, size );
MPI_Finalize();
return 0;
}
```

The MPI parallel environment


```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( "Hello world! I'm process %d of %d\n",rank, size );
MPI_Finalize();
return 0;
}
```

- How many is answered by a call to `MPI_Comm_size` as an `int` value,
- Who am I? Is answered by a call to `MPI_Comm_rank` as an `int` value that is conventionally called `rank` and is a number between `0` and `size-1`.

The MPI parallel environment

The last keyword we describe is the `MPI_COMM_WORLD`, this is the **Communicator object**.

Communicator

A **Communicator object** connects a group of processes in one MPI session. There can be more than one communicator in an MPI session, each of them gives each contained process an independent identifier and arranges its contained processes in an ordered topology.

This provides

- a **safe communication space**, that guarantees that the code can communicate as they need to, without conflicting with communication extraneous to the present code, e.g., if other parallel libraries are in use,
- a **unified object** for conveniently **denoting** communication context, the **group of communicating processes** and to house abstract process naming.

The MPI parallel environment

If we have saved our inquiring MPI program in the file `hamlet.c`, we can then modify our `Makefile` by modifying/adding the lines

```
all: helloworld hamlet
hamlet: hamlet.c
  $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
clean:
  rm -f helloworld hamlet
```

Then, we **compile everything** by doing `make hamlet` (or, simply, `make`).

The MPI parallel environment

If we have saved our inquiring MPI program in the file `hamlet.c`, we can then modify our `Makefile` by modifying/adding the lines

```
all: helloworld hamlet
hamlet: hamlet.c
  $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
clean:
  rm -f helloworld hamlet
```

Then, we **compile everything** by doing `make hamlet` (or, simply, `make`).

When we run the code with `mpirun -np 6 hamlet` we see

```
Hello world! I'm process 1 of 6
Hello world! I'm process 5 of 6
Hello world! I'm process 0 of 6
Hello world! I'm process 3 of 6
Hello world! I'm process 2 of 6
Hello world! I'm process 4 of 6
```

The MPI parallel environment

If we have saved our inquiring MPI program in the file `hamlet.c`, we can then modify our `Makefile` by modifying/adding the lines

```
all: helloworld hamlet
hamlet: hamlet.c
  $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
clean:
  rm -f helloworld hamlet
```

Then, we **compile everything** by doing `make hamlet` (or, simply, `make`).

When we run the code with `mpirun -np 6 hamlet` we see

```
Hello world! I'm process 1 of 6
Hello world! I'm process 5 of 6
Hello world! I'm process 0 of 6
Hello world! I'm process 3 of 6
Hello world! I'm process 2 of 6
Hello world! I'm process 4 of 6
```

- Every processor answers the call,

The MPI parallel environment

If we have saved our inquiring MPI program in the file `hamlet.c`, we can then modify our `Makefile` by modifying/adding the lines

```
all: helloworld hamlet
hamlet: hamlet.c
  $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
clean:
  rm -f helloworld hamlet
```

Then, we **compile everything** by doing `make hamlet` (or, simply, `make`).

When we run the code with `mpirun -np 6 hamlet` we see

```
Hello world! I'm process 1 of 6
Hello world! I'm process 5 of 6
Hello world! I'm process 0 of 6
Hello world! I'm process 3 of 6
Hello world! I'm process 2 of 6
Hello world! I'm process 4 of 6
```

- Every processor answers the call,
- But it answers it as soon as he has done doing the computation! There is no synchronization.

Update the repository

Now that we have another piece fo working code, we can update our *git* repository:

- We can run `git status` to see what we have changed and added,

Update the repository

Now that we have another piece fo working code, we can update our *git* repository:

- We can run `git status` to see what we have changed and added,
- Then we add the new file and the modified `Makefile` by doing:

  ```
  git add hamlet.c Makefile
  ```

Update the repository

Now that we have another piece fo working code, we can update our *git* repository:

- We can run `git status` to see what we have changed and added,
- Then we add the new file and the modified `Makefile` by doing:

  ```
  git add hamlet.c Makefile
  ```

- Now can *prepare our commit*:

  ```
  git commit -m "Test of MPI_Comm_rank/size functions"
  ```

Update the repository

Now that we have another piece fo working code, we can update our *git* repository:

- We can run `git status` to see what we have changed and added,
- Then we add the new file and the modified `Makefile` by doing:

  ```
  git add hamlet.c Makefile
  ```

- Now can *prepare our commit*:

  ```
  git commit -m "Test of MPI_Comm_rank/size functions"
  ```

- Finally we **push it** to the repository:

  ```
  git push
  ```

A word of advice

When should you **not** write parallel code with MPI?

- The **effort** of writing optimized and scalable MPI codes is **not negligible**, therefore a direct usage of it its usually best suited for developing *libraries for scientific computations*.

When should you write parallel code with MPI?

A word of advice

When should you **not** write parallel code with MPI?

- The **effort** of writing optimized and scalable MPI codes is **not negligible**, therefore a direct usage of it its usually best suited for developing *libraries for scientific computations*.
- If there is a library containing a good (possibly open source) parallel implementation of the algorithm and the data structure you need: **LEARN IT AND USE IT!**

When should you write parallel code with MPI?

A word of advice

When should you **not** write parallel code with MPI?

- The **effort** of writing optimized and scalable MPI codes is **not negligible**, therefore a direct usage of it its usually best suited for developing *libraries for scientific computations*.
- If there is a library containing a good (possibly open source) parallel implementation of the algorithm and the data structure you need: **LEARN IT AND USE IT!**

When should you write parallel code with MPI?

- When you are learning about parallel computing with distributed memory!

A word of advice

When should you **not** write parallel code with MPI?

- The **effort** of writing optimized and scalable MPI codes is **not negligible**, therefore a direct usage of it its usually best suited for developing *libraries for scientific computations*.
- If there is a library containing a good (possibly open source) parallel implementation of the algorithm and the data structure you need: **LEARN IT AND USE IT!**

When should you write parallel code with MPI?

- When you are learning about parallel computing with distributed memory!
- To *really* understand what the instructions manuals of such parallel libraries are telling you,

A word of advice

When should you **not** write parallel code with MPI?

- The **effort** of writing optimized and scalable MPI codes is **not negligible**, therefore a direct usage of it its usually best suited for developing *libraries for scientific computations*.
- If there is a library containing a good (possibly open source) parallel implementation of the algorithm and the data structure you need: **LEARN IT AND USE IT!**

When should you write parallel code with MPI?

- When you are learning about parallel computing with distributed memory!
- To *really* understand what the instructions manuals of such parallel libraries are telling you,
- Sometimes it happens, you are using a library based on MPI and some function that you truly need is not included.

A word of advice

When should you **not** write parallel code with MPI?

- The **effort** of writing optimized and scalable MPI codes is **not negligible**, therefore a direct usage of it its usually best suited for developing *libraries for scientific computations*.
- If there is a library containing a good (possibly open source) parallel implementation of the algorithm and the data structure you need: **LEARN IT AND USE IT!**

When should you write parallel code with MPI?

- When you are learning about parallel computing with distributed memory!
- To *really* understand what the instructions manuals of such parallel libraries are telling you,
- Sometimes it happens, you are using a library based on MPI and some function that you truly need is not included.
- To **develop** new and better **libraries** for your **scientific challenge!**