



High-Performance Mathematics

Programming in a distributed setting: MPI

Progetto Speciale per la Didattica 2023/24

Fabio Durastante (L2)

April 17, 2024





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
 - ▶ Point-to-Point Communications
 - Deadlock
 - Nonblocking communications
 - Sendreceive
 - Things left out
 - ▶ References



How do we realize practically this parallelism?

1 Distributed parallelism in practice

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?

1 Distributed parallelism in practice

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?

1 Distributed parallelism in practice

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?

1 Distributed parallelism in practice

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?

1 Distributed parallelism in practice

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
- ▶ Point-to-Point Communications
 - Deadlock
 - Nonblocking communications
 - Sendreceive
 - Things left out
- ▶ References



Message Passing Interface - www.mpi-forum.org

2 An Introduction to MPI

*“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

2 An Introduction to MPI

“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

2 An Introduction to MPI

“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

2 An Introduction to MPI

“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

2 An Introduction to MPI

“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 An Introduction to MPI

- 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

2 An Introduction to MPI

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 $Y = 1, 2, \dots$ 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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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](#)



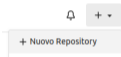
Putting up a git repository for our code

2 An Introduction to MPI

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](#)

3. Create a **new repository**:





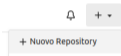
Putting up a git repository for our code

2 An Introduction to MPI

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](#)

3. Create a **new repository**:



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



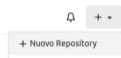
Putting up a git repository for our code

2 An Introduction to MPI

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](#)

3. Create a **new repository**:



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

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

Nome Repository *



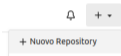
Putting up a git repository for our code

2 An Introduction to MPI

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](#)

3. Create a **new repository**:



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

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

Nome Repository *

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

`.gitignore`

Seleziona i template di `.gitignore`.



Putting up a git repository for our code

2 An Introduction to MPI

- 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

BSD License



Putting up a git repository for our code

2 An Introduction to MPI

- 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

BSD License

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



Putting up a git repository for our code

2 An Introduction to MPI

Nuovo Repository

Un repository contiene tutti i file del progetto, inclusa la cronologia delle revisioni. Lo hai già altrove? [Migrare il repository.](#)

Proprietario *

Alcune organizzazioni potrebbero non essere visualizzate nel menu a discesa a causa di un limite massimo al numero di repository.

Nome Repository *

Un buon nome per un repository è costituito da parole chiave corte, facili da ricordare e uniche.

Visibilità Rendi privato il repository
Solo il proprietario o i membri dell'organizzazione se hanno diritti, saranno in grado di vederlo.

Descrizione

Modello

Etichette Issue

.gitignore

Scegli di quali file non tenere traccia da un elenco di modelli per le lingue comuni. Gli artefatti tipici generati dagli strumenti di build di ogni lingua sono inclusi su .gitignore per impostazione predefinita.

Licenza

Una licenza governa ciò che gli altri possono e non possono fare con il tuo codice. Non sei sicuro di chi è giusto per il tuo progetto? Vedi [Scegli una licenza.](#)

LEGGIMI

Qui puoi scrivere una descrizione completa del progetto.

Inizializza Repository (Aggiungi .gitignore, Licenza e LEGGIMI)

Ramo (Branch) predefinito

Il ramo predefinito è il ramo base per le richieste di pull e i commit di codice.

Modello di Fiducia per la Firma

Seleziona il modello di fiducia per la verifica della firma. Le opzioni possibili sono:

- Collaboratore: Fidati delle firme da parte dei collaboratori
- Committer: Fidati delle firme che corrispondono ai committenti
- Collaboratore+Committer: Fidati delle firme da parte dei collaboratori che corrispondono al committer
- Predefinito: utilizzare il modello di trust predefinito per questa installazione

Modello Rendi il repository un modello

And then push:

Crea Repository



Putting up a git repository for our code

2 An Introduction to MPI

fdurastante/hpmcode

Non seguire più 1 Vota 0 Forka 0

Codice Problemi Pull Requests Pacchetti Progetti Rilasci Wiki Attività Impostazioni

Codici di esempio del progetto speciale High-Performance Mathematics.

Gestisci argomenti

1 Commit 1 Ramo (Branch) 0 Tag 27 KIB

main Vai al file Aggiungi file

HTTPS SSH git@git.phc.dm.unipi.it:fdurastante/hpmcode.git

File	Commit	Tempo
Fabio Durastante	cd58934e16 Initial commit	2 secondi fa
.gitignore	Initial commit	2 secondi fa
LICENSE	Initial commit	2 secondi fa
README.md	Initial commit	2 secondi fa

README.md

hpmcode

Codici di esempio del progetto speciale High-Performance Mathematics.

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
```

12:50 the link should be the one of your repository, not mine!



Hello (parallel) world!

2 An Introduction to MPI

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

```
#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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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 a 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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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?

```
#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

2 An Introduction to MPI

```
#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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

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

2 An Introduction to MPI

Now that we have another piece of 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

2 An Introduction to MPI

Now that we have another piece of 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

2 An Introduction to MPI

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

2 An Introduction to MPI

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 is usually best suited for developing *libraries for scientific computations*.

When should you write parallel code with MPI?



A word of advice

2 An Introduction to MPI

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 is 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

2 An Introduction to MPI

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 is 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

2 An Introduction to MPI

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 is 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

2 An Introduction to MPI

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 is 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

2 An Introduction to MPI

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 is 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!**



Table of Contents

3 Point-to-Point Communications

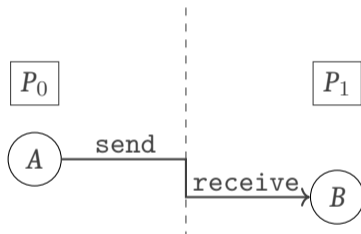
- ▶ Distributed parallelism in practice
- ▶ An Introduction to MPI
 - Preliminary work
 - Our First MPI Program
 - The MPI parallel environment
 - When to travel the MPI route
- ▶ **Point-to-Point Communications**
 - Deadlock
 - Nonblocking communications
 - Sendreceive
 - Things left out
- ▶ References



Sending and Receiving Messages

3 Point-to-Point Communications

We have seen that each process within a *communicator* is identified by its *rank*, how can we **exchange data** between two processes?



We need to possess several information to have a meaningful message

- Who is sending the data?
- To whom the data is sent?
- What type of data are we sending?
- How does the receiver can identify it?



The blocking send and receive

3 Point-to-Point Communications

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

void *message points to the message content itself, it can be a simple scalar or a group of data,

int count specifies the number of data elements of which the message is composed,
MPI_Datatype datatype indicates the **data type** of the elements that make up the message,

int dest the rank of the destination process,

int tag the user-defined tag field,

MPI_Comm comm the communicator in which the source and destination processes reside and for which their respective ranks are defined.



The blocking send and receive

3 Point-to-Point Communications

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

`void *message` points to the message content itself, it can be a simple scalar or a group of data,

`int count` specifies the number of data elements of which the message is composed, `MPI_Datatype datatype` indicates the **data type** of the elements that make up the message,

`int source` the rank of the source process,

`int tag` the user-defined tag field,

`MPI_Comm comm` the communicator in which the source and destination processes reside,

`MPI_Status *status` is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`.



Basic MPI Data Types

3 Point-to-Point Communications

Of the previous slides inputs the only ones that is specific to MPI is the MPI_Datatype:

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int

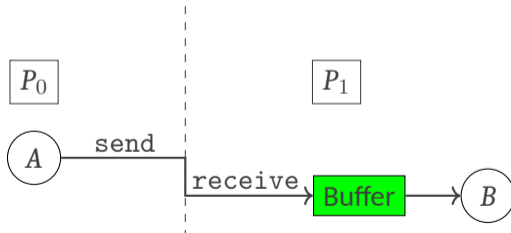


Why “blocking” send and receive?

3 Point-to-Point Communications

For the `MPI_Send` to be *locally blocking* means that it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer: it is a *non local* operation.

Note: The message might be copied directly into the matching receive buffer (as in the first figure), or it might be copied into a temporary system buffer.





Why “blocking” send and receive?

3 Point-to-Point Communications

For the `MPI_Send` to be *locally blocking* means that it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer: it is a *non local* operation.

The `MPI_Receive`, on the other hand returns **only** after the receive buffer contains the newly received message. A receive can't complete before the matching send has completed, but, of course, it can complete only after the matching send has started.



A simple send/receive example

3 Point-to-Point Communications

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

int main( int argc, char **argv){
char message[20]; int myrank; MPI_Status status;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
if (myrank == 0){ /* code for process zero */
strcpy(message, "Hello, there");
MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1){ /* code for process one */
MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
printf("received :%s:\n", message);
}
MPI_Finalize();
return 0; }
```




A simple send/receive example

3 Point-to-Point Communications

We can compile our code by simply adding to our Makefile

```
easysendrecv: easysendrecv.c
```

```
$(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
```

then, we type `make`, and we run our program with

```
mpirun -np 2 easysendrecv
```

getting as answer

```
received :Hello, there:
```

So, what have we done?



A simple send/receive example

3 Point-to-Point Communications

```
MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
```

Process 0 sends the content of the **char** array `message` [20], whose size is `strlen(message)+1` size of **char** (`MPI_CHAR`) to processor 1 with tag 99 on the communicator `MPI_COMM_WORLD`.

```
MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
```

on the other side process 1, receives into the buffer `message` [20] an array with size 20 size of `MPI_CHAR`, from process 0 with tag 99 on the same communicator `MPI_COMM_WORLD`.



A simple send/receive example : programmer smash!

3 Point-to-Point Communications

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?
- What happens if we have a mismatch in the ranks of the sending and receiving processes?
- What happens if we use the wrong message size?
- What happens if we have a mismatch in the type?



A simple send/receive example : programmer smash!

3 Point-to-Point Communications

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

- What happens if we use the wrong message size?

- What happens if we have a mismatch in the type?



A simple send/receive example : programmer smash!

3 Point-to-Point Communications

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

A: The process stays there hanging trying to match messages that will never come...

- What happens if we use the wrong message size?

- What happens if we have a mismatch in the type?



A simple send/receive example : programmer smash!

3 Point-to-Point Communications

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

A: The process stays there hanging trying to match messages that will never come...

- What happens if we use the wrong message size?

A: If the size of the arriving message is longer than the expected we get an error of `MPI_ERR_TRUNCATE: message truncated`, note that there are combinations of wrong sizes for which things still works

- What happens if we have a mismatch in the type?



A simple send/receive example : programmer smash!

3 Point-to-Point Communications

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

A: The process stays there hanging trying to match messages that will never come...

- What happens if we use the wrong message size?

A: If the size of the arriving message is longer than the expected we get an error of `MPI_ERR_TRUNCATE: message truncated`, note that there are combinations of wrong sizes for which things still works

- What happens if we have a mismatch in the type?

A: There are combinations of instances in which things seems to work, **but** the code is erroneous, and the behavior is not deterministic.



Checkpointing to git

3 Point-to-Point Communications

Exercise

It's a **good exercise** at this point to try updating your git repository with the new file and the updated Makefile.

Do you remember?

```
git status, git add, git commit - m "...", and then git push.
```




Checkpointing to git

3 Point-to-Point Communications


Exercise

It's a **good exercise** at this point to try updating your git repository with the new file and the updated Makefile.

Do you remember?

```
git status, git add, git commit - m "...", and then git push.
```

Exercise

A good idea as a home exercise is to try updating the README file as well. Inside you can use *Markdown* to format the text:  www.markdownguide.org.



Dealing with more than one send and receive

3 Point-to-Point Communications

We have two processes that exchange data: `MPI_Comm_rank(comm, &myrank);`

- Solution 1:

```
if (myrank == 0){
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
}else if(myrank == 1){
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
}
```



Dealing with more than one send and receive

3 Point-to-Point Communications

We have two processes that exchange data: `MPI_Comm_rank(comm, &myrank);`

- Solution 1:

```
if (myrank == 0){
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
}else if (myrank == 1){
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
}
```

- Solution 2:

```
if (myrank == 0){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
}else if (myrank == 1){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```



Dealing with more than one send and receive

3 Point-to-Point Communications

We have two processes that exchange data: `MPI_Comm_rank(comm, &myrank);`

- Solution 2:

```
if (myrank == 0){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
}else if (myrank == 1){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

- Solution 3:

```
if (myrank == 0){
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
}else if (myrank == 1){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```



Dealing with more than one send and receive

3 Point-to-Point Communications

In the case of Solution 1:

```
MPI_Comm_rank(comm, &myrank);  
if (myrank == 0){  
  MPI_Send(...);  
  MPI_Recv(...);  
}else if(myrank == 1){  
  MPI_Send(...);  
  MPI_Recv(...);  
}
```

- The call MPI_Send is blocking, therefore the message sent by each process has to be copied out before the send operation returns and the receive operation starts.
- For the call to complete successfully, it is then necessary that **at least one of the two messages sent be buffered**, otherwise ...
- a deadlock situation occurs: both processes are blocked since there is no buffer space available!



Dealing with more than one send and receive

3 Point-to-Point Communications

In the case of Solution 1:

```
MPI_Comm_rank(comm, &myrank);  
if (myrank == 0){  
    MPI_Send(...);  
    MPI_Recv(...);  
}else if(myrank == 1){  
    MPI_Send(...);  
    MPI_Recv(...);  
}
```

- The call MPI_Send is blocking, therefore the message sent by each process has to be copied out before the send operation returns and the receive operation starts.
- For the call to complete successfully, it is then necessary that **at least one of the two messages sent be buffered**, otherwise ...
- a deadlock situation occurs: both processes are blocked since there is no buffer space available!



Here what happens to your program when you encounter Deadlock



Dealing with more than one send and receive

3 Point-to-Point Communications

In the case of Solution 2:

```
MPI_Comm_rank(comm, &myrank);  
if (myrank == 0){  
    MPI_Recv(...);  
    MPI_Send(...);  
}else if(myrank == 1){  
    MPI_Recv(...);  
    MPI_Send(...);  
}
```

- The receive operation of process 0 must complete before its send. It can complete **only if** the matching send of processor 1 is executed.
- The receive operation of process 1 must complete before its send. It can complete **only if** the matching send of processor 0 is executed.
- This program will always deadlock.



Dealing with more than one send and receive

3 Point-to-Point Communications

In the case of Solution 2:

```
MPI_Comm_rank(comm, &myrank);  
if (myrank == 0){  
    MPI_Recv(...);  
    MPI_Send(...);  
}else if(myrank == 1){  
    MPI_Recv(...);  
    MPI_Send(...);  
}
```

- The receive operation of process 0 must complete before its send. It can complete **only if** the matching send of processor 1 is executed.
- The receive operation of process 1 must complete before its send. It can complete **only if** the matching send of processor 0 is executed.
- This program will always deadlock.



Here what happens to your program when you encounter Deadlock



Dealing with more than one send and receive

3 Point-to-Point Communications

In the case of Solution 3:

```
MPI_Comm_rank(comm, &myrank);  
if (myrank == 0){  
    MPI_Send(...);  
    MPI_Recv(...);  
}else if(myrank == 1){  
    MPI_Recv(...);  
    MPI_Send(...);  
}
```

- This program will succeed even if no buffer space for data is available.



Dealing with more than one send and receive

3 Point-to-Point Communications



This way you can beat
Deadlock!

In the case of Solution 3:

```
MPI_Comm_rank(comm, &myrank);  
if (myrank == 0){  
  MPI_Send(...);  
  MPI_Recv(...);  
}else if(myrank == 1){  
  MPI_Recv(...);  
  MPI_Send(...);  
}
```

- This program will succeed even if no buffer space for data is available.



Nonblocking communications

3 Point-to-Point Communications

As we have seen the use of **blocking communications** ensures that

- the send and receive buffers used in the `MPI_Send` and `MPI_Recv` arguments are safe to use or reuse after the function call,
- but it also means that unless there is a simultaneously matching send for each receive, the code will deadlock.



Nonblocking communications

3 Point-to-Point Communications

There exists a version of the point-to-point communication that **returns immediately** from the function call before confirming that the send or the receive has completed, these are the **nonblocking send** and **receive** functions.

- To verify that the data has been copied out of the send buffer a separate call is needed,
- To verify that the data has been received into the receive buffer a separate call is needed,



Nonblocking communications

3 Point-to-Point Communications

There exists a version of the point-to-point communication that **returns immediately** from the function call before confirming that the send or the receive has completed, these are the **nonblocking send** and **receive** functions.

- To verify that the data has been copied out of the send buffer a separate call is needed,
- To verify that the data has been received into the receive buffer a separate call is needed,
- The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.
- The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.



Nonblocking comms: MPI_Isend and MPI_Irecv

3 Point-to-Point Communications

The two nonblocking point-to-point communication call are then

```
int MPI_Isend(void *message, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *send_request);
```

```
int MPI_Irecv(void *message, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *recv_request);
```

- The MPI_Request variables substitute the MPI_Status and store information about the status of the pending communication operation.
- The way of saying when this communications **must** be completed is by using the `int MPI_Wait(MPI_Request *request, MPI_Status *status)` when is called, the nonblocking request originating from MPI_Isend or MPI_Irecv is provided as an argument.



Nonblocking communications: an example

3 Point-to-Point Communications

```
int main(int argc, char **argv) {
    int a, b, size, rank, tag = 0;
    MPI_Status status;
    MPI_Request send_request, recv_request;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        a = 314159;
        MPI_Isend(&a, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &send_request);
        MPI_Irecv (&b, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &recv_request);
        MPI_Wait(&send_request, &status);
        MPI_Wait(&recv_request, &status);
        printf ("Process %d received value %d\n", rank, b);
    }
}
```



Nonblocking communications: an example

continued

Continued from previous slide

```
else {  
a = 667;  
MPI_Isend (&a, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &send_request);  
MPI_Irecv (&b, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &recv_request);  
MPI_Wait(&send_request, &status);  
MPI_Wait(&recv_request, &status);  
printf ("Process %d received value %d\n", rank, b);  
}  
MPI_Finalize();  
return 0;  
}
```




A simple send/receive example

3 Point-to-Point Communications

We can compile our code by simply adding to our Makefile

```
nonblockingsendrecv: nonblockingsendrecv.c  
    $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
```

then, we type make, and we run our program with

```
mpirun -np 2 nonblockingsendrecv
```

getting as answer

```
Process 0 received value 667
```

```
Process 1 received value 314159
```



A simple send/receive example

3 Point-to-Point Communications

We can compile our code by simply adding to our Makefile

```
nonblockingsendrecv: nonblockingsendrecv.c  
    $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
```

then, we type make, and we run our program with

```
mpirun -np 2 nonblockingsendrecv
```

getting as answer

```
Process 0 received value 667  
Process 1 received value 314159
```

Another useful instruction for the case of nonblocking communication is represented by

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

A call to MPI_TEST returns `flag = true` if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation.



Send-Receive

3 Point-to-Point Communications

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process.

- Source and destination are possibly the same,
- Send-receive operation is very useful for executing a shift operation across a chain of processes,
- A message sent by a send-receive operation can be received by a regular receive operation

```
int MPI_Sendrecv(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int source,
int recvtag, MPI_Comm comm, MPI_Status *status);
```



Send-Receive-Replace

3 Point-to-Point Communications

A slight variant of the `MPI_Sendrecv` operation is represented by the `MPI_Sendrecv_replace` operation

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,  
int dest, int sendtag, int source, int recvtag, MPI_Comm comm,  
MPI_Status *status)
```

as the name suggests, the same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

Clearly, if you confront its arguments with the one of the `MPI_Sendrecv`, the arguments `void *recvbuf`, `int recvcount` are absent.



Things left out

3 Point-to-Point Communications

We are leaving out some variants of the point-to-point communication:

- Both for blocking and nonblocking communications we have left out the **synchronous** and **ready** mode,
- For nonblocking communications we have also the **buffered** variants,
- Instead of waiting/testing for a single communication at the time we could wait for the completion of some, or all the operations in a list. There are specific routines for achieving this.



You can read about this on the manual:

- [1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0. <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>, High Performance Computing Center Stuttgart (HLRS).



Table of Contents

4 References

- ▶ Distributed parallelism in practice
- ▶ An Introduction to MPI
 - Preliminary work
 - Our First MPI Program
 - The MPI parallel environment
 - When to travel the MPI route
- ▶ Point-to-Point Communications
 - Deadlock
 - Nonblocking communications
 - Sendreceive
 - Things left out
- ▶ **References**



References

4 References

There are more books, notes, tutorials, online courses and oral tradition on scientific and parallel computing than we would have time to read and listen in a life. Pretty much everything that contains the words Parallel Programming and Scientific Computing is good...

I suggest here the book

[1] Rouson, D., Xia, J., & Xu, X. (2011). Scientific software design: the object-oriented way. Cambridge University Press.

that discusses general aspect of scientific computing (not perfectly related to parallel computing), and to have on your bedside

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0. <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>, High Performance Computing Center Stuttgart (HLRS).