Calcolo Parallelo dall'Infrastruttura alla Matematica

An Introduction to MPI

Laurea Triennale e Magistrale in Matematica

Fabio Durastante

April 20, 2023

Dipartimento di Matematica
Università di Pisa

Table of Contents

1 An Introduction to MPI

- An Introduction to MPI
  - 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?

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.

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

1 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

CASSANDRA.
Troilus and Creſsida Act IIᵈ Scene 2ᵈ
Cry Trojans cry, practice your eyes with tears
Troy must not be nor goodly Ilion stand.
Published March 1, 1776 by J. Mortimer, Norfolk Street, Strand.

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

Hello (parallel) world!

1 An Introduction to MPI

In all the course we are going to use the MPI inside C programs.

```
#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!

Hello (parallel) world!

In all the course we are going to use the MPI inside C programs.

```
#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 now to *compile* and *link* the `helloworld.c` program, and we can do it simply by:

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

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

```
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

```
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.

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 something like

  ```
  icc (ICC) 17.0.4 20170411
  Copyright (C) 1985-2017 Intel Corporation.
  All rights reserved.
  ```

  for an Intel compiler.

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

```
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.

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

```
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.

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

```
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.

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.

“If you find yourself saying, ”But I don’t want to use wrapper compilers!”, please humor us and try them. See if they work for you. Be sure to let us know if they do not work for you. ” -

`https://www.open-mpi.org/faq/?category=mpi-apps`

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

```c
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.

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

1 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

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

1 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

1 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

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.

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!**

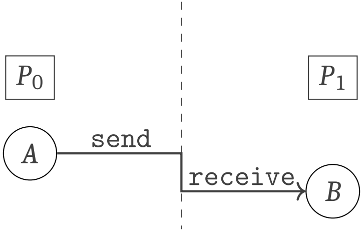
Table of Contents

2 Point-to-Point Communications

- An Introduction to MPI
  - 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

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 posses 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

```
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

```
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

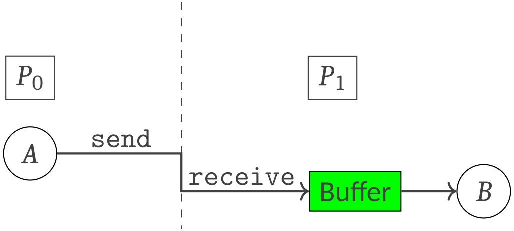
Of the previous slides inputs the only ones that is specific to MPI is the `MPI_Datatype`:

`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?

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?

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

```c
#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

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

```
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!

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!

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!

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!

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!

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.

Dealing with more than one send and receive

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

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

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

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

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

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

Here what happens to your program when you encounter Deadlock

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

2 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

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

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

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

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

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

```
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);
}
```

Continue on the next slide

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

2 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

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

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

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

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

3 References

- An Introduction to MPI
  - 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

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).

Calcolo Parallelo dall'Infrastruttura alla Matematica *Thank you for listening!*

Any questions?