Calcolo Parallelo : Lezione 1

Fabio Durastante

Dipartimento di Matematica, Università di Pisa

Master in Scienze e Tecnologie Spaziali, 2022

1. Scientific and parallel computing
   - Flynn's Taxonomy
   - The TOP500 List
2. Parallel Algorithms
3. Message Passing Interface
   - Our First MPI Program
   - The MPI parallel environment
   - When to travek the MPI road
4. Point-to-Point Communications
   - Deadlock
   - Nonblocking communications
   - Sendreceive
   - Things left out
5. References

Scientific and parallel computing

"**Computational science** (also **scientific computing** or **scientific computation** (SC)) is a rapidly growing multidisciplinary field that uses advanced computing capabilities to *understand and solve complex problems*. It is an area of science which spans many disciplines, but at its core it involves the development of *models and simulations to understand natural systems*."

Wikipedia

What are the applications?

- Computational finance,
- Computational biology,
- Simulation of complex systems,
- Network analysis
- Multi-physics simulations,
- Weather and climate models,
- ...

What are the applications?

- Computational finance,
- Computational biology,
- Simulation of complex systems,
- Network analysis
- Multi-physics simulations,
- Weather and climate models,
- ...

Why the need for parallelism?

Moore's law

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."

G. Moore, 1975

Moore's law

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."

G. Moore, 1975

Computers *should* reach the physical limits of Moore's Law at some point in the 2020s...exponential functions saturates physical capabilities!

Motivations for parallel computing

- We are hitting the wall of single processor transistor count/computing capabilities,

Motivations for parallel computing

- We are hitting the wall of single processor transistor count/computing capabilities,
- Some applications needs more memory than the one that could be available on a single machine,

Motivations for parallel computing

- We are hitting the wall of single processor transistor count/computing capabilities,
- Some applications needs more memory than the one that could be available on a single machine,
- Optimization of sequential algorithms can bring us only to a certain extent

Motivations for parallel computing

- We are hitting the wall of single processor transistor count/computing capabilities,
- Some applications needs more memory than the one that could be available on a single machine,
- Optimization of sequential algorithms can bring us only to a certain extent

"διαίρει καὶ βασίλευε"
(diáirei kái basíleue)

Motivations for parallel computing

- We are hitting the wall of single processor transistor count/computing capabilities,
- Some applications needs more memory than the one that could be available on a single machine,
- Optimization of sequential algorithms can bring us only to a certain extent

"διαίρει καὶ βασίλευε"
(diáirei kái basíleue)
Dividi et Impera

Motivations for parallel computing

- We are hitting the wall of single processor transistor count/computing capabilities,
- Some applications needs more memory than the one that could be available on a single machine,
- Optimization of sequential algorithms can bring us only to a certain extent

"διαίρει καὶ βασίλευε"
(diáirei kái basíleue)
Dividi et Impera

Therefore, we need

- Algorithms that can work in parallel,
- A communications protocol for parallel computation integrated with our programming languages
- Parallel machines that can actually run this code

Parallel computers: Flynn's Taxonomy

Let us start from the bottom: the machines.

Parallel computers: Flynn's Taxonomy

Let us start from the bottom: the machines.

- What is a parallel computer?

Parallel computers: Flynn's Taxonomy

Let us start from the bottom: the machines.

- What is a parallel computer? well, it can be a certain number of different "things"
  - Multi-core computing
  - Symmetric multiprocessing
  - Distributed computing
  - Cluster computing
  - Massively parallel computing
  - Grid computing
  - General-purpose computing on graphics processing units (GPGPU)
  - Vector processors

Parallel computers: Flynn's Taxonomy

Let us start from the bottom: the machines.

- What is a parallel computer? well, it can be a certain number of different "things"
  - Multi-core computing
  - Symmetric multiprocessing
  - Distributed computing
  - Cluster computing
  - Massively parallel computing
  - Grid computing
  - General-purpose computing on graphics processing units (GPGPU)
  - Vector processors

Parallel computers: Flynn's Taxonomy

Let us start from the bottom: the machines.

- What is a parallel computer?
- Let us *abstract* from the machine by describing Flynn's taxonomy

Single instruction stream, single data stream
SISD

Single instruction stream, multiple data streams
SIMD

Multiple instruction streams, single data stream
MISD

Multiple instruction streams, multiple data streams
MIMD

Parallel computers: Flynn's Taxonomy

Let us start from the bottom: the machines.

- What is a parallel computer?
- Let us *abstract* from the machine by describing Flynn's taxonomy

Single instruction stream, single data stream
SISD

Single instruction stream, multiple data streams
SIMD

Multiple instruction streams, single data stream
MISD

Multiple instruction streams, multiple data streams
MIMD

Parallel Computers: our computer model

For our task of introducing parallel computations we need to fix a **specific multiprocessor model**, i.e., a specific generalization of the sequential RAM model in which there is more than one processor.

Since we want to stay in a SIMD/MIMD model, we focus on a *local memory machine model*, i.e., a set of M processors each with its own local memory that are attached to a common communication network.

Parallel Computers: our computer model

For our task of introducing parallel computations we need to fix a **specific multiprocessor model**, i.e., a specific generalization of the sequential RAM model in which there is more than one processor.

Since we want to stay in a SIMD/MIMD model, we focus on a *local memory machine model*, i.e., a set of M processors each with its own local memory that are attached to a common communication network.

- We can be more precise about the connection between processors, one can consider a network (a collection of switches connected by communication channels) and delve in a detailed way into its pattern of interconnection, i.e., into what is called the network topology.

Parallel Computers: our computer model

For our task of introducing parallel computations we need to fix a **specific multiprocessor model**, i.e., a specific generalization of the sequential RAM model in which there is more than one processor.

Since we want to stay in a SIMD/MIMD model, we focus on a *local memory machine model*, i.e., a set of M processors each with its own local memory that are attached to a common communication network.

- An alternative is to summarize the network properties in terms of two parameters: latency and bandwidth

  Latency the time it takes for a message to traverse the network;

  Bandwidth the rate at which a processor can inject data into the network.

The TOP500 List – https://www.top500.org/

"...we have decided in 1993 to assemble and maintain a list of the 500 most powerful computer systems. Our list has been compiled twice a year since June 1993 with the help of high-performance computer experts, computational scientists, manufacturers, and the Internet community in general...
In the present list (which we call the TOP500), we list computers ranked by their performance on the LINPACK Benchmark."
http://www.netlib.org/benchmark/hpl/

The TOP500 List – https://www.top500.org/

"...we have decided in 1993 to assemble and maintain a list of the 500 most powerful computer systems. Our list has been compiled twice a year since June 1993 with the help of high-performance computer experts, computational scientists, manufacturers, and the Internet community in general...
In the present list (which we call the TOP500), we list computers ranked by their performance on the LINPACK Benchmark."
http://www.netlib.org/benchmark/hpl/

The LINPACK Benchmark.
Solution of a dense $n \times n$ system of linear equations $A\mathbf{x} = \mathbf{b}$, so that

- $\frac{\|A\mathbf{x}-\mathbf{b}\|}{\|A\|\|\mathbf{x}\|n\varepsilon} \leq O(1)$, for ε machine precision,
- It uses a specialized right–looking LU factorization with look–ahead
- Measuring
  - R_{max} the performance in GFLOPS for the largest problem run on a machine,
  - N_{max} the size of the largest problem run on a machine,
  - $N_{1/2}$ the size where half the R_{max} execution rate is achieved,
  - R_{peak} the theoretical peak performance GFLOPS for the machine.

Parallel Algorithms

In a fairly general way we can say that a **parallel algorithm** is an algorithm which can do *multiple operations* in a given time.

Parallel Algorithms

In a fairly general way we can say that a **parallel algorithm** is an algorithm which can do *multiple operations* in a given time.
Example: the sum of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$
\begin{aligned}
\mathbf{x} \quad &= [x_1 \; x_2 \; \cdots \; x_i \quad x_{i+1} \; \cdots x_n] \\
+ \\
\mathbf{y} \quad &= [y_1 \; y_2 \; \cdots \; y_i \quad y_{i+1} \; \cdots y_n] \\
= \\
\mathbf{x} + \mathbf{y} \quad &= [x_1 + y_1 \; x_2 + y_2 \; \cdots \; x_i + y_i \quad \cdots x_n + y_n]
\end{aligned}
$$

- If we do the operation sequentially we do $O(n)$ operations in T_n

Parallel Algorithms

In a fairly general way we can say that a **parallel algorithm** is an algorithm which can do *multiple operations* in a given time.
Example: the sum of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$
\begin{aligned}
\mathbf{x} \quad &= [x_1 \; x_2 \; \cdots \; x_i \mid x_{i+1} \; \cdots x_n] \\
+ \quad & \\
\mathbf{y} \quad &= [y_1 \; y_2 \; \cdots \; y_i \mid y_{i+1} \; \cdots y_n] \\
= \quad & \\
\mathbf{x} + \mathbf{y} \quad &= [x_1 + y_1 \; x_2 + y_2 \; \cdots \; x_i + y_i \mid \cdots x_n + y_n]
\end{aligned}
$$

- If we do the operation sequentially we do $O(n)$ operations in T_n
- If we split the operation among 2 processors, one summing up the entries between $1, \ldots, i$, and one summing up the entries between $i+1, \ldots, n$ we take T_i time for the first part and T_{n-i} time for the second, therefore the overall time is $\max(T_i, T_{n-i})$ for doing always $O(n)$ operations.

Parallel Algorithms: *speedup*

Let us try to think again in an abstract way and to quantify the overall speed gain for a given gain in a subset of a process.

- We break some process into N distinct portions with the ith portion occupying the P_i fraction of the overall completion time,

Parallel Algorithms: *speedup*

Let us try to think again in an abstract way and to quantify the overall speed gain for a given gain in a subset of a process.

- We break some process into N distinct portions with the ith portion occupying the P_i fraction of the overall completion time,
- then we order such portion in such a way that the Nth portions subsumes all the parts of the overall processes that have fixed costs.

Parallel Algorithms: *speedup*

Let us try to think again in an abstract way and to quantify the overall speed gain for a given gain in a subset of a process.

- We break some process into N distinct portions with the ith portion occupying the P_i fraction of the overall completion time,
- then we order such portion in such a way that the Nth portions subsumes all the parts of the overall processes that have fixed costs.
- The *speedup* of the ith portion can then be defined as

$$S_i \triangleq \frac{t_{\text{original}}}{t_{\text{optimized}}}, \quad i = 1, \ldots, N-1$$

  where the numerator and denominator are the original and optimized completion time.

Parallel Algorithms: *speedup*

Let us try to think again in an abstract way and to quantify the overall speed gain for a given gain in a subset of a process.

- We break some process into N distinct portions with the ith portion occupying the P_i fraction of the overall completion time,
- then we order such portion in such a way that the Nth portions subsumes all the parts of the overall processes that have fixed costs.
- The *speedup* of the ith portion can then be defined as

$$S_i \triangleq \frac{t_{\text{original}}}{t_{\text{optimized}}}, \quad i = 1, \ldots, N-1$$

  where the numerator and denominator are the original and optimized completion time.

Amdahl's Law

Then the overall speedup for $\mathbf{P} = (P_1, \ldots, P_N)$, $\mathbf{S} = (S_1, \ldots, S_{N-1})$ is:

$$S(\mathbf{P}, \mathbf{S}) = \left(P_N + \sum_{i=1}^{N-1} \frac{P_i}{S_i} \right)^{-1}.$$

Parallel Algorithms: *Amdahl's Law*

Let us make some observations on Amdahl's Law

- We are not assuming about whether the original completion time involves some optimization,
- We are not making any assumption on what our optimization process is,
- We are not even saying that the process in question involves a computer!

Amdahl's Law is a fairly general way of looking at how processes can be speed up by dividing them into sub-tasks with lower execution time.

Parallel Algorithms: *Amdahl's Law*

Let us make some observations on Amdahl's Law

- We are not assuming about whether the original completion time involves some optimization,
- We are not making any assumption on what our optimization process is,
- We are not even saying that the process in question involves a computer!

Amdahl's Law is a fairly general way of looking at how processes can be speed up by dividing them into sub-tasks with lower execution time. Moreover, it fixes the theoretical maximum speedup in various scenarios.

- If we allow all components S_i to grow unbounded then the upper bound on all scenario si $S_{\text{max}} = 1/P_N$.

Let us decline it in the context of the potential utility of *parallel hardware*.

Parallel Algorithms: *Amdahl's Law* for *parallel hardware*

Consider now having a parallel machine that permits us dividing the execution of code across M hardware units, then the problem independent maximum speedup that such hardware can provide is M.

Parallel Efficiency

We define the parallel efficiency E as

$$E \triangleq \frac{S_{\text{overall}}}{M},$$

where $E = 100\%$ correspond to the maximal use of the available hardware. When $S_{\text{max}} < M$, it is then impossible to take full advantage of all available execution units.

Parallel Algorithms: *Amdahl's Law* for *parallel hardware*

Consider now having a parallel machine that permits us dividing the execution of code across M hardware units, then the problem independent maximum speedup that such hardware can provide is M.

Parallel Efficiency

We define the parallel efficiency E as

$$E \triangleq \frac{S_{\text{overall}}}{M},$$

where $E = 100\%$ correspond to the maximal use of the available hardware. When $S_{\text{max}} < M$, it is then impossible to take full advantage of all available execution units.

Goal: we require very large S_{max} and correspondingly tiny P_N.

Parallel Algorithms: *Amdahl's Law* for *parallel hardware*

Consider now having a parallel machine that permits us dividing the execution of code across M hardware units, then the problem independent maximum speedup that such hardware can provide is M.

Parallel Efficiency

We define the parallel efficiency E as

$$E \triangleq \frac{S_{\text{overall}}}{M},$$

where $E = 100\%$ correspond to the maximal use of the available hardware. When $S_{\text{max}} < M$, it is then impossible to take full advantage of all available execution units.

Goal: we require very large S_{max} and correspondingly tiny P_N.

<u>Every dusty corner of a code must scale</u>, any portion that doesn't becomes the rate-limiting step!

Parallel Algorithms: *Amdahl's Law* limitations

What we are neglecting and what we are tacitly assuming

- We are neglecting *overhead costs*, i.e., the cost associated with parallel execution such as
  - initializing (spawning) and joining of different computation threads,
  - communication between processes, data movement and memory allocation.
- We considered also the ideal case in which $S_i \to +\infty \ \forall i$, observe that with finite speedup on portions 1 through $N-1$, the S_{overall} might continue to improve with increasing number of execution units.
- We are assuming that the size of the problem remains fixed while the number of execution units increases, this is called the case of strong scalability. In some contexts, we need to turn instead to weak scalability in which the problem size grows proportionally to the number of execution units.

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.

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 – https://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 – https://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 – https://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 – https://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 – https://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

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!

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 Open MPI implementation of MPI.
- 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 Open MPI implementation of MPI.
- 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 Open MPI implementation of MPI.
- 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 Open MPI implementation of MPI.
- 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 Open MPI implementation of MPI.
- 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`[1], 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 very 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
```

[1] https://www.gnu.org/software/make/

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`.

Queue management systems

To run on a shared machine you will most certainly encounter a *so-called* queue management system/workload manager, usually one between

- SLURM slurm.schedmd.com
- PBS www.openpbs.org

They are "fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters.", meaning:

- You don't simply run `mpirun`/`mpiexec` asking for resources on a shared machine, with many users chaos would immediately follow,
- You **ask for resources** for your **job** and put it into an **execution queue**.
- When the resources you need will be available your program will be executed (for the requested amount of time)

On the machine we use for this course we have SLURM installed, so we are going to give usage details only about it.

SLURM: basic usage

The key idea is that we produce a bash script containing the information on what we want to be executed and give it to SLURM to put into the queue.

We create a file `run-helloworld.sh` with

```
#!/bin/bash
#SBATCH --job-name=helloworld      # Job name (can be called with %x)
#SBATCH --partition=Prod           # Node partition: here only Prod
#SBATCH --nodes=1                  # Run all processes on a single node
#SBATCH --ntasks=6                 # Run ntasks processes (-np ntasks)
#SBATCH --time=00:05:00            # Time limit hrs:min:sec
#SBATCH -o ./%x.%j.out             # Standard Output (%j job number)
#SBATCH -e ./%x.%j.err             # Standard Error  (%j job number)
mpirun helloworld
```

Then we put it into the queue by doing

```
sbatch run-helloworld.sh
```

To see if the **job** has been put into the queue (or executed) you can use the command `squeue`, if you are on a machine that has many user and a long queue you may need to trim down the list by doing, e.g., `squeue | grep parallel_job` that will search in the list just the job with your name.

SLURM: basic usage

There are (a lot) more options that can be used in SLURM, not all of them make sense for all programming models.
Other commands that are useful are:

`sinfo` reports the state of partitions and nodes managed by Slurm. It has a wide variety of filtering, sorting, and formatting options.

`scancel` is used to cancel a pending or running job or job step. It can also be used to send an arbitrary signal to all processes associated with a running job or job step.

Important. If you request a combination of resources that does not make sense for SLURM, e.g., more nodes than you have, wrong partition, more time than possible, a non-matching set of nodes and tasks, then the job will fail on execution. This means that you may end up waiting lot of time to be executed, and then fail with no answer whatsoever. Some unreasonable combination (e.g., unknown partition) may by noticed by `sbatch`, but don't count on it.

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

If we now run

```
sbatch run-helloworld.sh
```

we get a `helloworld.xx.out` file containing the following output:

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

Every process executes the line

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

that it is a local routine!

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

If we now run

```
sbatch run-helloworld.sh
```

we get a `helloworld.xx.out` file containing the following output:

```
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 vs 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?

```
#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

```
#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 need to describe is the `MPI_COMM_WORLD`, this is the standard Communicator object.

Communicator

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

This provides

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

The MPI parallel environment

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

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

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

The MPI parallel environment

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

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

Then, we compile everything by doing `make hamlet` (or, simply, `make`). When we run the code with `mpirun -np 6 hamlet` we see

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

The MPI parallel environment

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

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

Then, we compile everything by doing `make hamlet` (or, simply, `make`). When we run the code with `mpirun -np 6 hamlet` we see

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

- Every processor answers the call,

The MPI parallel environment

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

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

Then, we compile everything by doing `make hamlet` (or, simply, `make`). When we run the code with `mpirun -np 6 hamlet` we see

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

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

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!

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`.

Why "blocking" send and receive?

For the `MPI_Send` to be 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 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

```
#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?

```
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 now two processes that needs to exchange some data

- Solution 1:

```
MPI_Comm_rank(comm, &myrank);
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 now two processes that needs to exchange some data

- Solution 1:

```
MPI_Comm_rank(comm, &myrank);
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:

```
MPI_Comm_rank(comm, &myrank);
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 now two processes that needs to exchange some data

- Solution 2:

```
MPI_Comm_rank(comm, &myrank);
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:

```
MPI_Comm_rank(comm, &myrank);
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

Here what happens to your program when you encounter Deadlock

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 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

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.

Deadlock Issues

We can try to salvage what the situation in the case of Solution 1 by allocating the buffer space for the send calls

```
if (myrank == 0){
 MPI_Send(...);
 MPI_Recv(...);
}else if(myrank == 1){
 MPI_Send(...);
 MPI_Recv(...);
}
```

We can substitute the MPI_Send operation with a Send in buffered mode

```
int MPI_Bsend(const void* buf, int count,
MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

- A buffered mode send operation can be started whether or not a matching receive has been posted;
- It may complete before a matching receive is posted;
- This operation is *local*!

Deadlock Issues

We can try to salvage what the situation in the case of Solution 1 by allocating the buffer space for the send calls

We can substitute the MPI_Send operation with a Send in buffered mode

```
int MPI_Bsend(const void* buf, int count,
MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

```
if (myrank == 0){
 MPI_Send(...);
 MPI_Recv(...);
}else if(myrank == 1){
 MPI_Send(...);
 MPI_Recv(...);
}
```

- A buffered mode send operation can be started whether or not a matching receive has been posted;
- It may complete before a matching receive is posted;
- This operation is *local*!
- The bad news is that if the buffer space is not enough we have exchanged the deadlock error with a buffer overflow condition;

Deadlock Issues

We can try to salvage what the situation in the case of Solution 1 by allocating the buffer space for the send calls

```
if (myrank == 0){
 MPI_Send(...);
 MPI_Recv(...);
}else if(myrank == 1){
 MPI_Send(...);
 MPI_Recv(...);
}
```

We can substitute the MPI_Send operation with a Send in buffered mode

```
int MPI_Bsend(const void* buf, int count,
MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

- A buffered mode send operation can be started whether or not a matching receive has been posted;
- It may complete before a matching receive is posted;
- This operation is *local*!
- The good news is that buffer overflow condition are easier to detect than deadlock.

Allocating buffer space

To actually use the `MPI_Bsend` we need also to allocate the space for the buffer, therefore we need to use the two functions

```
int MPI_Buffer_attach(void* buffer, int size)
int MPI_Buffer_detach(void* buffer_addr, int* size)
```

- `MPI_Buffer_attach` provides a buffer in the user's memory to be used for buffering outgoing messages, where `buffer` is the starting address of a memory region
- `MPI_Buffer_detach` detaches the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted.

Allocating buffer space

To actually use the MPI_Bsend we need also to allocate the space for the buffer, therefore we need to use the two functions

```
int MPI_Buffer_attach(void* buffer, int size)
```

```
int MPI_Buffer_detach(void* buffer_addr, int* size)
```

```
#define BUFFSIZE 10000
int size; char *buff;
// Buffer of 10000 bytes for MPI_Bsend
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
// Buffer size reduced to zero
MPI_Buffer_detach( &buff, &size);
// Buffer of 10000 bytes available again
MPI_Buffer_attach( buff, size);
```

Note: a pointer to the buffer is passed to MPI_Buffer_attach while the address of the pointer is passed to MPI_Buffer_detach and these are both `void *`.

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.

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.

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.

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

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.

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 communications: 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);
} 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

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 from this presentation 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:

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).

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

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

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).