Parallel Scientific Computing

Course AMS301 — Fall 2023 — Lecture 1

Context, motivation and generalities
Parallel architectures, algorithms and programming

Axel Modave

Context, motivation and generalities

Parallel architectures and algorithms

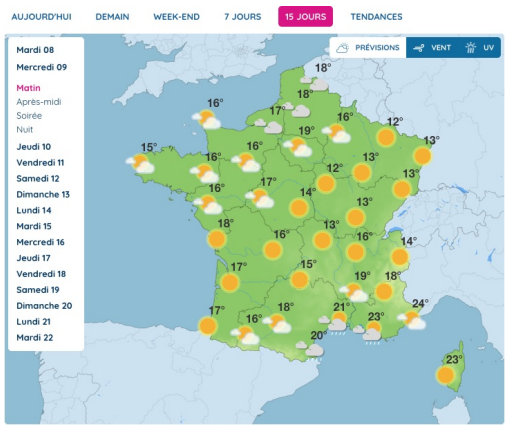
Parallel programming with MPI in C++

Parallel computing — Context and motivation [1/2]

We consider large-scale problems that are difficult or even impossible to solve with standard computers . . .

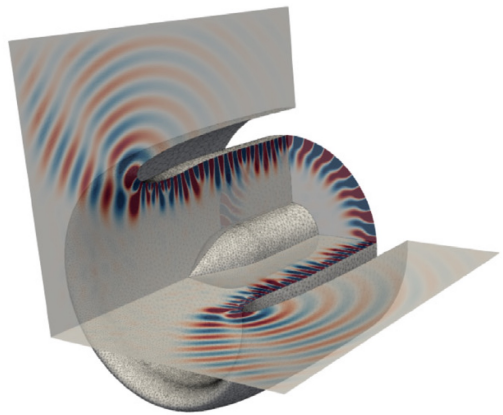
- because the **computation time** is too long,
- because the **amount of data** to be stored is too large.

Examples of large-scale problems



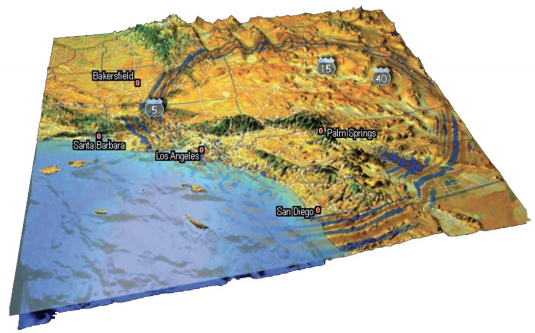
Weather forecast

(Picture Météo France)



Simulation R&D
Aeronautic industry

(Picture Siemens Indus. Soft.)



Simulation of
earthquake

(Picture SPECFEM3D)

Parallel computing — Context and motivation [2/2]

Parallel computing allows for . . .

- using **more computing power** $\longrightarrow$ *Decreasing the computation time*
- using **more memory space** $\longrightarrow$ *Processing a larger amount of data*

Examples of parallel computers

Standard supercomputer

- Composed of several nodes
- Each node: processor(s) + RAM *(Random Memory Access)*
- Nodes connected with an internal network

Cluster of standard computers

- Composed of several machines
- Each machine: processor(s) + RAM
- Machines connected with a standard network *(e.g. Ethernet/Wi-Fi)*

Parallel computing — What is different? [1/3]

Parallel architectures

Using a **parallel machine** is more complicated than using a standard computer.

A parallel machine is composed of . . .

- **several nodes** *(each of them with dedicated computing/memory resources)*,
- an **interconnection network**.

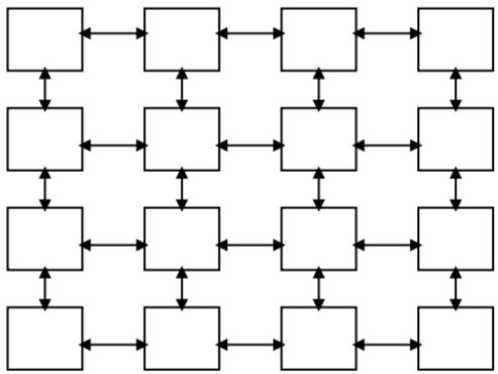


Illustration of a standard supercomputer

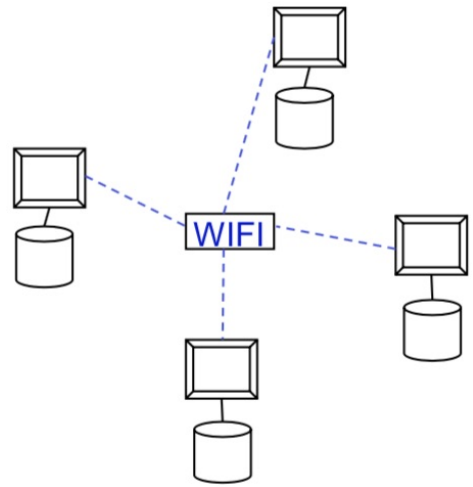
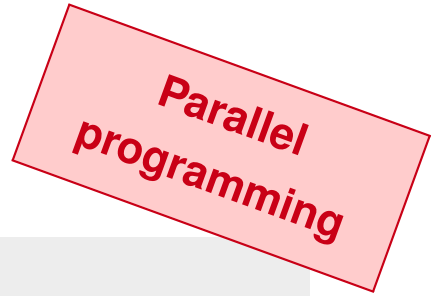


Illustration of a cluster of standard computers

To run a program on these machines, we have to specify . . .

- The resources to be used *(How many nodes? Which nodes?)*;
- The distribution of operations and data on the nodes;
- How to manage dependencies between the operations?

Parallel computing — What is different? [2/3]



To control the parallel execution of a program, we have to:

- modify the **code** *(additional variables and functions)*,
- use a *"parallel"* **compiler** *(or a standard compiler with a parallel library)*,
- add options at the **execution**.

The `MPI` **library** *(Message Passing Interface)* gives the functions required to manage data transfers between `MPI` processes *("from one node to another one")*.

Example of `MPI` functions in `C++` to send an integer from one process to another one:

```
MPI_Send(&value, 1, MPI_INT, idTo, 0, MPI_COMM_WORLD)
MPI_Recv(&value, 1, MPI_INT, idFrom, 0, MPI_COMM_WORLD, status)
```

To compile, we use the *"parallel version"* of standard compilers:

`gcc` $\longrightarrow$ `mpicc`

`g++` $\longrightarrow$ `mpicxx`

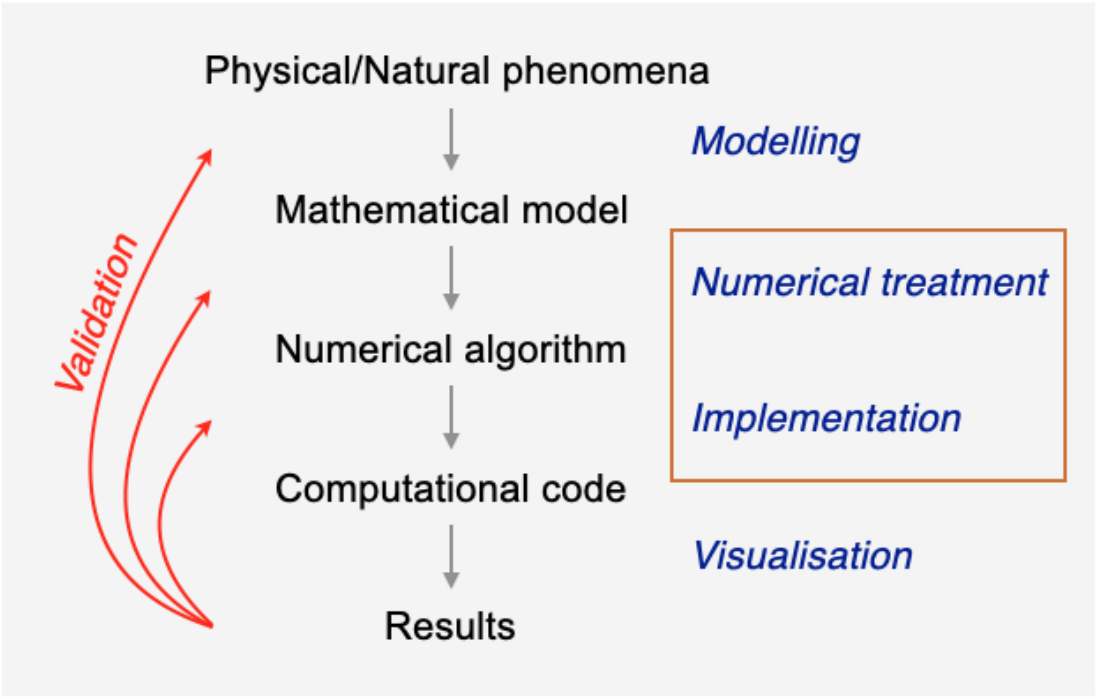
To execute, we use the command `mpirun` to give the execution options.

Parallel computing — What is different? [3/3]

Parallel algorithms

- For the parallel numerical solution of scientific problems, the codes must be parallelized *(i.e. parallel treatment of operations/data)*.
- In order to *real*ly leverage the computational power of the parallel machines, we have to **rethink the numerical methods for parallel computing** to take into account the characteristics of the parallel architectures.

Design of a numerical simulation tool



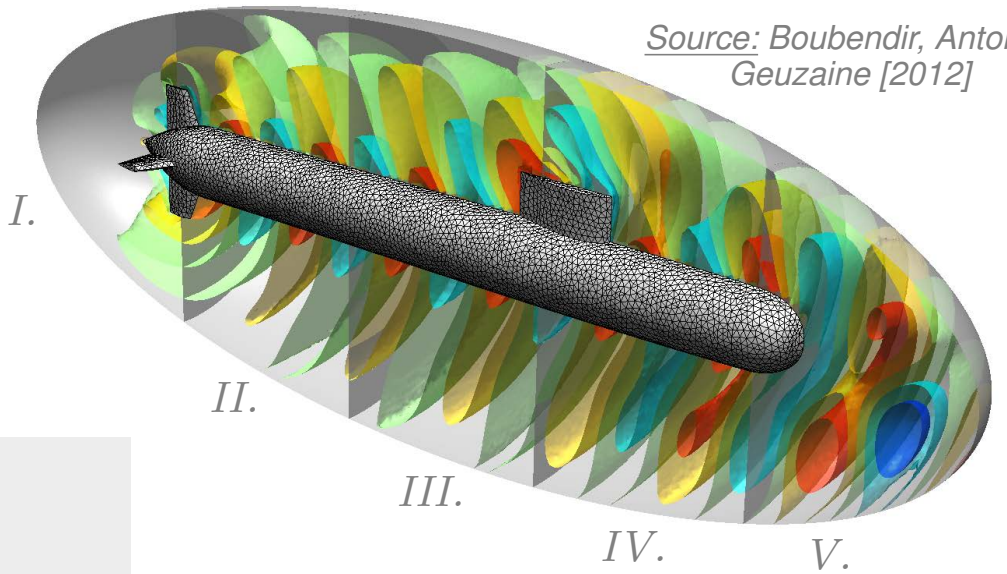
Parallel computing — Example of a parallel numerical method

Finite element solution with a domain decomposition method (DDM)

Find $u \in H^1(\Omega)$ s.t. $(\nabla u, \nabla v)_\Omega - (k^2 u, v)_\Omega + \langle \imath k u, v \rangle_{\partial\Omega} = (f, v)_\Omega, \forall v \in H^1(\Omega)$

Standard direct/iterative procedure:

Large system to solve;
Parallel strategies not efficient
or slow convergence



Source: Boubendir, Antoine, Geuzaine [2012]

Decomposition of the initial problem into subproblems solved in parallel in an iterative process

Procedure with DDM:

Small linear system to solve

$$\begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \begin{bmatrix} x \\ x \end{bmatrix} = \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

for each subdomain

Data transfers at the interfaces

For each Ω_i, find $u_i \in H^1(\Omega_i)$ s.t. $(\nabla u_i, \nabla v_i)_{\Omega_i} - (k^2 u_i, v_i)_{\Omega_i} + \sum_{\text{interfaces}}$??? $\cdots$

AMS301 course [1/3]

Goals of AMS301 { Taking into account the parallel aspects in the design and the implementation of numerical methods for efficient simulations

Numerical and algorithmic aspects of parallel computing

Course proposed in the following programs:

- 3A *"Modélisation et simulation"* (parcours ModSim) at ENSTA Paris
- 3A *"Mathématiques pour la santé et l'environnement"* at ENSTA Paris
- M2 *"Mathématiques et Applications"* (parcours AMS) at IP Paris and UPSay
- M2 *"Informatique"* (parcours HPDA) at IP Paris

Other courses of 3A ModSim and M2 AMS:

- Focused on programming aspects:
  - AMS-O12 – Cours accéléré de programmation *(Bloc 0)*
    `https://pmarchand.pages.math.cnrs.fr/slides/courses/master_AMS_O12/`
  - AMS-I03 – Programmation hybride et multi-cœurs *(Bloc 2)*
    `https://perso.ensta-paris.fr/~tajchman/`
- Focused on numerical aspects:
  - AMS-X02 – Méthodes num. avancées et calcul haute performance *(Bloc 2)*

AMS301 course [2/3]

Goals of AMS301

Taking into account the parallel aspects in the design and the implementation of numerical methods for efficient simulations

Numerical and algorithmic aspects of parallel computing

Content

(approx. 1h lecture and 2h exercices per session)

- Paradigms and fundamentals of parallel scientific computing
- Parallel solution of linear algebraic systems
- Parallel solution of partial differential problems *(with finite differences/elements)*
- Parallel programming with MPI in C++

Prerequisite knowledge and skills

- Basic knowledge on num. linear algebra and num. methods for PDE problems
- Basic knowledge on the UNIX environment and the C++ language

https://ams301.pages.math.cnrs.fr/

AMS301 course [3/3]

Evaluation *No written exam!*

- Two programming project *(evaluation based on written reports and C++ codes)*
- Oral presentation for the second project, with questions related to the lectures

Axel Modave
Chargé de recherche CNRS
UMA / ENSTA Paris
Office 22.29

Nicolas Kielbasiewicz
Ingénieur de recherche CNRS
UMA / ENSTA Paris
Office 22.16

https://ams301.pages.math.cnrs.fr/

Context, motivation and generalities

Parallel architectures and algorithms

Parallel programming with MPI in C++

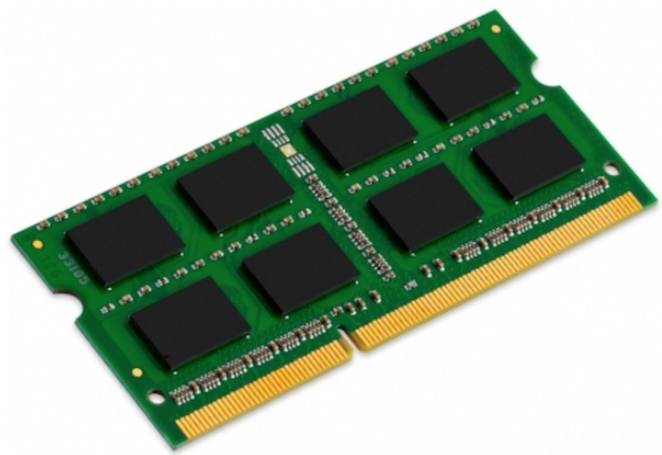
Working stations at ENSTA Paris

Architectures — Standard computer [2/3]

Standard computers are composed of . . .

- **processing units**:
  - one processor *(CPU — Central Processing Unit)*
  - one graphic card *(GPU — Graphical Processing Unit)*
- **memory units**:
  - fast memory *(RAM — Random Access Memory)* 7.75 GB
  - one hard drive *(HDD — Hard Disk Drive)* ∼ 1 TB
- **connections** for data transfers:
  - between HDD and RAM
  - between RAM and the *“cache”* memory of processing units

Intel Core i5 processor
Real size: 3.75cm x 3.75cm



RAM

Performance of working stations at ENSTA Paris

	# Cores	Frequency	Arithmetic throughput
CPU : Intel(R) Core(TM) i5-4430	4	3.0 GHz	$\sim$ 15 GFLOP/s *(double precision?)*
GPU : ATI Mobility Radeon HD 5430	80	675 MHz	$\sim$ 80 GFLOP/s *(single precision)*

	Memory size
Fast memory (RAM)	7.75 GB
Hard drive (HDD)	$\sim$ 1 TB

	Memory bandwidth
PCI express	$\sim$ 10 GB/s

Warning: These numbers are approximate, but the orders should be correct.

10^3 kilo (k)	10^{12} tera (T)
10^6 mega (M)	10^{15} peta (P)
10^9 giga (G)	10^{18} exa (E)

Arithmetic throughput $\approx$ "Rate of computing"

FLOP = Floating-Point Operation

Memory size

byte (B) = 8 bits $(bit = binary\ digit)$

Storage of a float with single precision = 4 bytes
Storage of a float with double precision = 8 bytes

Memory bandwidth $\approx$ "Rate of data transfer"

Architectures — Standard supercomputer [1/3]

General view

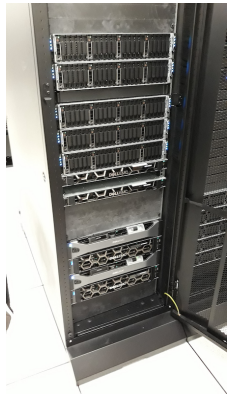
Front

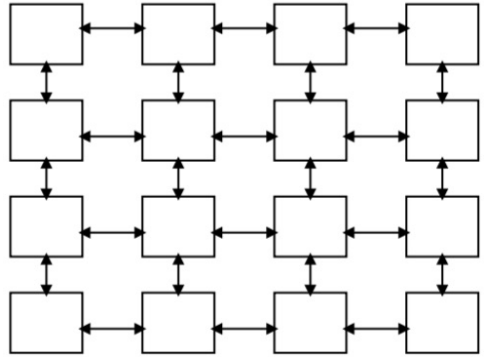
Back

Cluster `Cholesky` *— IDCS mesocentre — IP Paris*

`http://meso-ipp.gitlab.labos.polytechnique.fr/user_doc/`

Architectures — Standard supercomputer [2/3]



The cluster `Cholesky` is composed of ...

- *2 login front-end nodes*
- **processing units**:
  - 68 *"CPU nodes"* with 2 CPUs
  - 4 *"GPU nodes"* with 2 CPUs and 4 GPUs
- **memory units**:
  - one fast memory *(RAM)* on each node *(private)* — 160 to 384 GB/node
  - one parallel file system *(shared)* — 385 TB
- **connections for data transfers**:
  - between RAM and the *"cache"* memory of processing units on each node
  - between nodes (RAM) and file system *(interconnection network)*

`http://meso-ipp.gitlab.labos.polytechnique.fr/user_doc/`

Architectures — Standard supercomputer [3/3]

Performance of cluster `Cholesky`

	# Cores	Frequency	Arithmetic throughput	Quantity
CPU : Intel Xeon CPU Gold 6230	20	2.1 GHz	$\sim$ 1 715 GFLOPS/s	**144**
GPU : Nvidia V100	5 120		$\sim$ 14 TFLOP/s	**8**
GPU : Nvidia A100	6 912		$\sim$ 19 TFLOP/s	**8**

	Memory size *(one node)*	Memory size *(32 nodes)*
Fast memory (RAM)	160 to 384 GB	$\sim$ 16 TB
Hard drive (HDD)		385 TB

😀 *Computational power* ↗
$\sim$ 0.5 PFLOP/s

😀 *Memory size* ↗

	Memory bandwidth
Transfers in the node (PCI express)	$\sim$ 48 GB/s
Interconnection network (InfiniBand)	100 GB/s

😡 *Transfers between nodes*

Warning: These numbers are approximate,
but the orders should be correct.

10^3 kilo (k)	10^{12} tera (T)
10^6 mega (M)	10^{15} peta (P)
10^9 giga (G)	10^{18} exa (E)

Architectures — World #3 supercomputer (June 1, 2021)

Cluster LUMI *(EUROHPC/CSC, Kajaani, Finland)*

Nodes: 1 536 CPU nodes + 2 560 GPU nodes
Processor: AMD EPYC 64C 2GHz *(2 per CPUnode)*
GPU: AMD MI250X GPUs *(4 per GPU node)*
Total peak performance: 214.35 PFLOP/s

`https://www.lumi-supercomputer.eu/`

`https://www.top500.org/`

Parallel algorithms

The design of parallel algorithms is more complicated
than the design of sequential algorithms.

- To design a **sequential algorithm**, we have to
  - define a sequence of instructions to be process in a particular order by a sequential machine.

  ... that's it. :-)

- To design a **parallel algorithm**, we have to
  - distribute the operations/data between the nodes of the parallel machine, *(each node has its own sequence of instructions and its own data)*
  - specify the data transfers between the nodes,
  - specify the order of operations.

Parallel algorithms — Addition of two vectors $\mathbf{z} = \mathbf{x} + \mathbf{y}$

Sequential algorithm

for $n = 0 \cdots 99$ **do**
$\quad z_n = x_n + y_n$
end

Parallel algorithm with 2 processes

Data: each process knows half of the entries of $\mathbf{x}$ and $\mathbf{y}$.

On each process $p = 0, 1$ **:**

for $n = (50 \cdot p) \cdots (50 \cdot (p+1) - 1)$ **do**
$\quad z_n = x_n + y_n$
end

Result: each process knows half of the entries of $\mathbf{z}$.

A **process** is a set of instructions, a memory space and resources for in/out operations.

With the ***"divide and conquer"*** strategy, the problem is divided into smaller problems that are distributed between the processes.

Parallel algorithms — Scalar product $S = \mathbf{x} \cdot \mathbf{y}$

Sequential algorithm

for $n = 0 \cdots 99$ **do**
$\quad S = S + x_n \cdot y_n$
end

Parallel algorithms with 2 processes

Data: each process knows half of the entries of $\mathbf{x}$ et $\mathbf{y}$.

On each process $p = 0, 1$**:**

for $n = (50 \cdot p) \cdots (50 \cdot (p+1) - 1)$ **do**
$\quad S^{(p)} = S^{(p)} + x_n \cdot y_n$
end

Communication: process 1 sends $S^{(1)}$ to process 0.

On process 0**:** $S = S^{(0)} + S^{(1)}$

Result: process 0 knows S.

A **synchronization** is introduced by the communication.

Parallel algorithms *that are far less basic*

Some important fields:

1. Numerical linear algebra *(with dense and sparse matrices)*

$$\text{Find } \mathbf{x} \in \mathbb{R}^N, \text{ such that } \mathbf{A}\mathbf{x} = \mathbf{b}.$$

2. Problems with structured grids *(ex. différence finies)*

$$\frac{u_{i,j}^{\ell+1} - u_{i,j}^{\ell}}{\Delta t} + \frac{u_{i+1,j}^{\ell} + u_{i-1,j}^{\ell} + u_{i,j+1}^{\ell} + u_{i,j-1}^{\ell} - 4u_{i,j}^{\ell}}{\Delta x^2} = 0$$

3. Problems with unstructured *(ex. éléments finis)*

$$M_{ij} = \int_{\Omega} \psi_i(\mathbf{x})\, \psi_j(\mathbf{x})\, d\mathbf{x} \qquad K_{ij} = \int_{\Omega} \nabla\psi_i(\mathbf{x}) \cdot \nabla\psi_j(\mathbf{x})\, d\mathbf{x}$$

4. Spectal methods

$$\text{FFT: } \hat{f}_n = \sum_{m=0}^{N-1} f_m\, e^{-2\pi\imath mn/N} \quad (n = 0 \ldots N-1)$$

5. $N-$body problems, Monte Carlo methods, ...
6. Algorithms on graphs *(e.g. in operational research)*

To be continued during AMS301 ...

Context, motivation and generalities

Parallel architectures and algorithms

Parallel programming with MPI in C++

MPI — Introduction [1/3]

The MPI standard

- MPI *(“Message Passing Interface”)* is an **coding standard** to program applications and libraries for computing on parallel architectures:
  - Supercomputers *(with an internal network – e.g. Infiniband)*
  - Clusters of computers *(with an external network – e.g. Ethernet/WIFI)*
  - Standard computers ... with one processor, one RAM and one hard drive
- MPI defines the syntax and semantics of library routines for writing portable message-passing programs in `C`, `C++`, and `Fortran`.
- The MPI specifications are decided by a consortium, the `MPI Forum`, composed of academics, laboratories and companies *(Intel, Cray, ATOS, Microsoft ...)*.

`https://www.mpi-forum.org/`

History

Year	Version	Specifications	
1991			Start of discussions for a new standard
1994	MPI 1.0	236 pages	MPI 1.1 (1995), MPI 1.2 (2008)
1997	MPI 2.0	370 pages	MPI 2.1 (2008), MPI 2.2 (2009)
2012	MPI 3.0	852 pages	MPI 3.1 (2015)
2021	MPI 4.0	1139 pages	

MPI libraries and installation

- Several implementations for `Fortran` and/or `C`/`C++`:
  - OpenMPI *(open source)*
  - MPICH *(open source)*
  - ...
- Installation possible via depots:
  - On Linux, using `apt-get`:

    ```
    >> sudo apt-get install libopenmpi-dev    (OpenMPI)
    >> sudo apt-get install mpich             (MPICH)
    ```

  - On macOS, using `macport`:

    ```
    >> sudo port install openmpi              (OpenMPI)
    >> sudo port install mpich                (MPICH)
    ```

How to use MPI?

- **Library with functions, constants and data types**

  The header of the C++ library must be included:

  ```
  #include <mpi.h>
  ```

  Functions allow for actions during the execution and/or they give informations:

  ```
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nbTask);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Send(...);
  MPI_Recv(...);
  MPI_Finalize();
  ```

- **Compilation with the MPI library**

  ```
  >> mpicxx myCode.cpp
  ```

  This is the standard compiler (g++ here) with the options to include MPI.

- **Environment for parallel execution**

  ```
  >> mpirun -np 4 a.out
  ```

  `mpirun` is used to run the program in parallel, with options to give informations on the parallel execution *(e.g. number of parallel processes)*.

MPI — Example: `Hello, World!`

Code

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);             // Initialize MPI
  cout << "Hello, World!" << endl; // Every proc prints the message
  MPI_Finalize();                     // Finalize MPI
  return 0;
}
```

Compilation and execution

```
>> mpicxx helloworld.cpp
>> mpirun -np 3 a.out
Hello, World!
Hello, World!
Hello, World!
```

The program must include:

- The header file `mpi.h`
- `MPI_Init(...)` before the first call to a MPI function
- `MPI_Finalize()` after the last call to a MPI function

The number of MPI processes is chosen at the execution *(here, 3)*. It cannot be modified.

MPI — Example: I Am Number Four

Code

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                    // Initialize MPI
  int nbTask;
  int myRank;
  MPI_Comm_size(MPI_COMM_WORLD, &nbTask);    // Get total nb of proc
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);    // Get rank for each proc
  cout << "I am task " << myRank << " out of " << nbTask << endl;
  MPI_Finalize();                            // Finalize MPI
  return 0;
}
```

Compilation and execution

```
>> mpicxx numberfour.cpp
>> mpirun -np 4 a.out
I am task 2 out of 4
I am task 0 out of 4
I am task 3 out of 4
I am task 1 out of 4
```

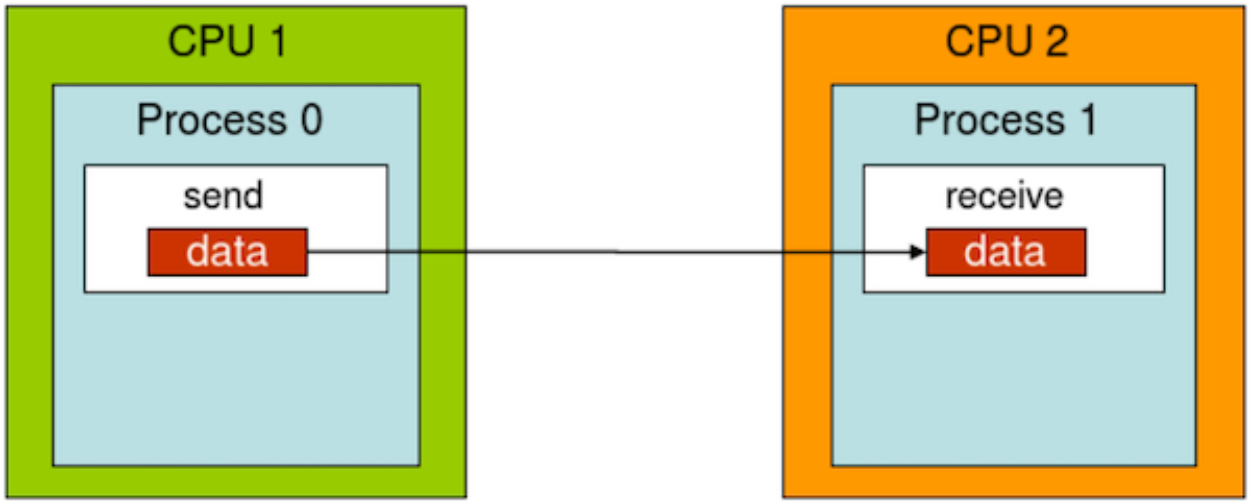
`MPI_Comm_size(...)` and `MPI_Comm_rank(...)` allow to get the total number of MPI processes, and the rank of the current one *(here, from 0 to 3)*.

The rank can be used to differentiate the work to be performed by each MPI process.

$\Longleftarrow$ The order of display cannot be predicted.

MPI — Point-to-point communication with `MPI_Send/MPI_Recv` [1/2]

The routines for point-to-point communications, `MPI_Send(...)` and `MPI_Recv(...)`, allow for the transfer of data from one given process to another one.



Example

```
1 if(myRank == 0)
2   MPI_Send(arraySend, 8, MPI_INT, 1, 666, MPI_COMM_WORLD);
3 if(myRank == 1)
4   MPI_Recv(arrayRecv, 8, MPI_INT, 0, 666, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

In this example, an array of 8 integers is sent from process `0` to process `1`.
The tag associated to the message is 666.

The functions are **blocking**: while the message is not totally received by process `1`, both processes (`0` and `1`) are blocked. ⇒ **WARNING !!!**

MPI — Point-to-point communication with `MPI_Send/MPI_Recv` [2/2]

The entries of the functions give informations on the data and the transfer:

```
int MPI_Send(const void*  buf,      Pointer to data to send
             int          count,    Size of the array to send
             MPI_Datatype datatype, Type of data to send
             int          dest,     Rank of the process "destination"
             int          tag,      Tag of the message
             MPI_Comm     comm)     Communicator

int MPI_Recv(void*        buf,      Pointer to storage for received data
             int          count,    Size of the array to receive
             MPI_Datatype datatype, Type of data to receive
             int          source,   Rank of the process "source"
             int          tag,      Tag of the message
             MPI_Comm     comm,     Communicator
             MPI_Status*  status)   Status of the communication
```

Data type: `MPI_INT` (`int`), `MPI_FLOAT` (`float`), `MPI_DOUBLE` (`double`), ...

To transfer a message, the `tag` must be identical in the `send` and in the `recv`.
Any number can be use.

The communicator specify the group of MPI processes in which the comunication is performed. During this course, only `MPI_COMM_WORLD` shall be used.

MPI — Example: `Sending arrays and vectors`

Sending a **single** `double` number:

```
1 double data = 3.14185;
2 MPI_Send(&data, 1, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
```

Sending a **static array** of `double` numbers:

```
1 double data[5];
2 data[0] = 0.1; data[1] = 9.9; data[2] = 0; data[3] = 2; data[4] = 4;
3 MPI_Send(data, 5, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
```

Sending a **vector** of `double` numbers:

```
1 vector<double> data(5);
2 data[0] = 0.1; data[1] = 9.9; data[2] = 0; data[3] = 2; data[4] = 4;
3 MPI_Send(&data[0], 5, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
```

. . . or only the two last values . . .

```
1 vector<double> data(5);
2 data[0] = 0.1; data[1] = 9.9; data[2] = 0; data[3] = 2; data[4] = 4;
3 MPI_Send(&data[3], 2, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
```

MPI — Example: Deadlock

The functions `MPI_Send` and `MPI_Recv` are **blocking**: while the message is not completely received, both processes involved in the communication are blocked.

Version 1

```
1 if(myRank == 0){
2   MPI_Recv(b, 100, MPI_DOUBLE, 1, 39, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3   MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);
4 }
5 if(myRank == 1){
6   MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7   MPI_Send(a, 100, MPI_DOUBLE, 0, 39, MPI_COMM_WORLD);
8 }
```

Both processes are blocked in the “send” mode, mutually waiting themself.
The program is blocked. $\Longrightarrow$ **Deadlock :-(**

Version 2

```
1 if(myRank == 0){
2   MPI_Recv(b, 100, MPI_DOUBLE, 1, 39, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3   MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);
4 }
5 if(myRank == 1){
6   MPI_Send(a, 100, MPI_DOUBLE, 0, 39, MPI_COMM_WORLD);
7   MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
8 }
```

Everything is alright. $\Longrightarrow$ **Happy face :-)**

MPI — Example: Addition of the N first integers

```
1  int main(int argc, char *argv[]){
2    int myrank, np;
3    MPI_Init(&argc, &argv);
4    MPI_Comm_size(MPI_COMM_WORLD, &np);
5    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6
7    int N        = 1000;
8    int startval = N * myrank / np + 1
9    int endval   = N * (myrank+1) / np
10   int partSum  = 0;
11
12   for(int i=startval; i<=endval; i++)
13     partSum += i ;
14   cout << "Partial sum on proc " << myrank << " equals " << partSum << endl ;
15
16   if(myrank != 0)
17     MPI_Send(&partSum, 1, MPI_INT, 0, 23, MPI_COMM_WORLD ) ;
18   else{
19     for(int j=1; j<np; j++) {
20       int tmp = 0;
21       MPI_Recv(&tmp, 1, MPI_INT, j, 23, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
22       partSum += tmp ;
23     }
24     cout << "The sum from 1 to " << N << " is: " << partSum << endl;
25   }
26
27   MPI_Finalize();
28 }
```

Summary

- **Parallel architectures**
  - Processing units: CPU, GPU
  - Memory units: HDD, RAM, *“cache”* memory
  - Supercomputer – Compute node – Hybrid machine
  - Interconnection network
- **Parallel algorithms**
  - Distributed operations and data
  - Process, task and data transfer
  - *“Divide and Conquer”* strategy
- **Basic MPI Commands**
  - `MPI_Init`
  - `MPI_Finalize`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send`
  - `MPI_Recv`