The individual power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 3GHz.
The individual power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 3GHz.

The recent evolution of the processors goes in the direction of several hearts (several computing units inside a single chip)
Generalities

- The individual power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 3GHz.
- The recent evolution of the processors goes in the direction of several hearts (several computing units inside a single chip)
- Scientific programs are sometimes calculations that can be performed independently for some parts and therefore use this available power is very wise
The individual power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 3GHz.

The recent evolution of the processors goes in the direction of several hearts (several computing units inside a single chip)

Scientific programs are sometimes calculations that can be performed independently for some parts and therefore use this available power is very wise

There are two main methods for making a program usable by many compute units: MPI and OPENMP
Use a library that will manage the exchange of information (and data) between computing units and thus structure the program according to the distribution of the calculations on available units and exchanges of data between them. The reference library is MPI (Message Passing Interface).
Use a library that will manage the exchange of information (and data) between computing units and thus structure the program according to the distribution of the calculations on available units and exchanges of data between them. The reference library is MPI (Message Passing Interface).

This library and the associated environment allows you to run a program (which is different on each unit of calculation) and which can transmit in a simple way information to the other computing units which are identified from the launch of the program.
Use a library that will manage the exchange of information (and data) between computing units and thus structure the program according to the distribution of the calculations on available units and exchanges of data between them. The reference library is MPI (Message Passing Interface).

This library and the associated environment allows you to run a program (which is different on each unit of calculation) and which can transmit in a simple way information to the other computing units which are identified from the launch of the program.

This method is the most general because it allows to do it on virtual computing units and on different machines connected by a network (fast if possible). The second method that applies exclusively to a single machine with a common memory and several calculation units. The library is OPENMP.
Outline

- MPI: definition and different versions
Outline

- MPI: definition and different versions
- Definition of the world of communicator, initialization and end of execution
Outline

- MPI: definition and different versions
- Definition of the world of communicator, initialization and end of execution
- Compiling and running a parallel program
Outline

- MPI: definition and different versions
- Definition of the world of communicator, initialization and end of execution
- Compiling and running a parallel program
- Physical limitations of communications
MPI: definition and different versions
Definition of the world of communicator, initialization and end of execution
Compiling and running a parallel program
Physical limitations of communications
Message sending: point-to-point communication
Outline

- MPI: definition and different versions
- Definition of the world of communicator, initialization and end of execution
- Compiling and running a parallel program
- Physical limitations of communications
- Message sending: point-to-point communication
- Message sending: broadcast, scatter and gather
Outline

- MPI: definition and different versions
- Definition of the world of communicator, initialization and end of execution
- Compiling and running a parallel program
- Physical limitations of communications
- Message sending: point-to-point communication
- Message sending: broadcast, scatter and gather
- Reduction
MPI: definition et different versions

MPI: definition et different versions

- MPiCH version 2 (contains communications across many physical media (Ethernet, Infiniband, Myrinet) exists on many distributions.
MPI: definition et different versions

- MPiCH version 2 (contains communications across many physical media (Ethernet, Infiniband, Myrinet) exists on many distributions.
- OPENMPI contains many physical media. It is in the form of a compiled package for the GNU compiler. Version 2.1 on an Ubuntu version 18.04
Definition of the communicator, initialization and end of execution
Definition of the communicator, initialization and end of execution (2)
#include<stdio.h>
#include<mpi.h>

char processor_name[MPI_MAX_PROCESSOR_NAME];
int numprocs,myid,namelen;
int main (int argc, char *argv[])
{
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    printf("Hello, Process %d of %d on %s\n", myid, numprocs, processor_name);

    MPI_Barrier( MPI_COMM_WORLD);
    MPI_Finalize();
}
Definition of the communicator, initialization and end of execution (4)

- the MPI_Init function initializes the world of units that will communicate
the MPI_Init function initializes the world of units that will communicate

The MPI_Finalize function destroys the world of units that have communicated
the MPI_Init function initializes the world of units that will communicate.

The MPI_Finalize function destroys the world of units that have communicated.

the MPI_BARRIER function is a signal that allows all compute units to be synchronized.
Definition of the communicator, initialization and end of execution

- The MPI_Init function initializes the world of units that will communicate.
- The MPI_Finalize function destroys the world of units that have communicated.
- The MPI_Barrier function is a signal that allows all compute units to be synchronized.
- The numprocs value retrieves the number of units chosen at runtime.
Definition of the communicator, initialization and end of execution

- the MPI_Init function initializes the world of units that will communicate
- The MPI_Finalize function destroys the world of units that have communicated
- the MPI_BARRIER function is a signal that allows all compute units to be synchronized
- numprocs value retrieves the number of units chosen at runtime
- processor_name is the string corresponding to the name of the compute unit.
Definition of the communicator, initialization and end of execution

- the MPI_Init function initializes the world of units that will communicate
- The MPI_Finalize function destroys the world of units that have communicated
- the MPI_Barrier function is a signal that allows all compute units to be synchronized
- numprocs value retrieves the number of units chosen at runtime
- processor_name is the string corresponding to the name of the compute unit.
- the integer values myid and namelen retrieve the number of the current compute unit and the string size from associated characters as the name of the calculation unit
Compiling and running a parallel code

- For compiling the aforementioned code hello.c, write in a terminal window:
  `mpicc hello.c -O3 -o hello`

- For running a parallel code, write in a terminal window:
  `mpirun -np 8 hello`

In practice, it is better not to exceed a number of units higher than the number of hearts of the machine.
Compiling and running a parallel code

- For compiling the aforementioned code hello.c, write in a terminal window:
  \texttt{mpicc hello.c -O3 -o hello}

- For running a parallel code, write in a terminal window:
  \texttt{mpirun -np 8 hello}
Compiling and running a parallel code

- For compiling the aforementioned code hello.c, write in a terminal window
  `mpicc hello.c -O3 -o hello`

- For running a parallel code, write in a terminal window
  `mpirun -np 8 hello`

- You can choose the number of units you want, because the calculation units are virtual. The operating system fits your request with the physical units. In practice, it is better not to exceed a number of units higher than the number of hearts of the machine.
To exchange information between computing units, the library has specific instructions to send messages.
To exchange information between computing units, the library has specific instructions to send messages.

These messages are not instantly transmitted to other calculation units. To start a communication passing for example by a network card, the response time is greater than several $\mu$s. The transfer speed is limited network capacity (e.g. 1Gb/s) and is much lower than what happens inside a computing unit.
To exchange information between computing units, the library has specific instructions to send messages.

These messages are not instantly transmitted to other calculation units. To start a communication passing for example by a network card, the response time is greater than several $\mu s$. The transfer speed is limited network capacity (e.g. 1Gb/s) and is much lower than what happens inside a computing unit.

You can send messages either between two calculation units (point-to-point message) or globally (e.g. the first unit sends to all others or all send information to all. In the latter case, we can observe significant performance falls.
```c
#include<stdio.h>
#include<mpi.h>
char processor_name[MPI_MAX_PROCESSOR_NAME];
int numprocs,myid,namelen;
MPI_Status status;
int main (int argc, char *argv[]) {
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    int entier=myid;
    printf("Hello, Process %d of %d on %s valeur de entier %d\n", myid, numprocs, processor_name,entier);
    if(myid==0) {
      MPI_Send(&entier,1,MPI_INT,3,10,MPI_COMM_WORLD);
    }
    if(myid==3) {
      MPI_Recv(&entier,1,MPI_INT,0,10,MPI_COMM_WORLD,&status);
    }
    MPI_Barrier( MPI_COMM_WORLD);
    printf("Hello, Process %d of %d on %s valeur de entier %d\n", myid, numprocs, processor_name,entier);
    MPI_Finalize();
}
```
To send data between two units: MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);
To send data between two units: `MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);`

- First argument: address of the variable

- Second argument: number of elements: 1 by default, otherwise the number of elements of the array,

- Third argument: the type of the variable, here `MPI_INT`,

- Fourth argument: the compute unit that receives the message.

- Fifth argument: a mark characterizing the message: the 10 value can be changed but it must keep the same for the reception.

- Sixth argument: the MPI communicator `MPI_COMM_WORLD`
To send data between two units: `MPI_Send(& integer, 1, MPI_INT, 3, 10, MPI_COMM_WORLD);`

- First argument: address of the variable
- Second argument: number of elements: 1 by default, otherwise the number of elements of the array,
To send data between two units: MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);

- First argument: address of the variable
- Second argument: number of elements: 1 by default, otherwise the number of elements of the array,
- Third argument: the type of the variable, here MPI_INT,
To send data between two units: `MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);

- First argument: address of the variable
- Second argument: number of elements: 1 by default, otherwise the number of elements of the array,
- Third argument: the type of the variable, here `MPI_INT`,
- Fourth argument: the compute unit that receives the message.
To send data between two units: MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);

- First argument: address of the variable
- Second argument: number of elements: 1 by default, otherwise the number of elements of the array,
- Third argument: the type of the variable, here MPI_INT,
- Fourth argument: the compute unit that receives the message.
- Fifth argument: a mark characterizing the message: the 10 value can be changed but it must keep the same for the reception.
To send data between two units: `MPI_Send(& integer, 1, MPI_INT, 3, 10, MPI_COMM_WORLD);

First argument: address of the variable

Second argument: number of elements: 1 by default, otherwise the number of elements of the array,

Third argument: the type of the variable, here `MPI_INT`,

Fourth argument: the compute unit that receives the message.

Fifth argument: a mark characterizing the message: the 10 value can be changed but it must keep the same for the reception.

Sixth argument: the MPI communicator `MPI_COMM_WORLD`
Message sending: point-to-point communication (3)

To receive data between two units: MPI_Recv (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);
To receive data between two units: `MPI_Recv (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);`

First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory
To receive data between two units: MPI_Recv (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);

First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory

Second argument: number of elements: 1 by default, otherwise the number of elements of the array,
To receive data between two units: `MPI_Recv (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);

First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory

Second argument: number of elements: 1 by default, otherwise the number of elements of the array,

Third argument: the type of the variable, here MPI_INT,
To receive data between two units: MPI_Recv (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);

First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory

Second argument: number of elements: 1 by default, otherwise the number of elements of the array,

Third argument: the type of the variable, here MPI_INT,

Fourth argument: the computing unit that sends the message.
To receive data between two units: MPI_Recv (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);

First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory

Second argument: number of elements: 1 by default, otherwise the number of elements of the array,

Third argument: the type of the variable, here MPI_INT,

Fourth argument: the computing unit that sends the message.

Fifth argument: a mark characterizing the message: the value 10 can be changed but it must keep the same for the reception.
To receive data between two units: `MPI_Recv (& integer, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, & status);`

- First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory)
- Second argument: number of elements: 1 by default, otherwise the number of elements of the array,
- Third argument: the type of the variable, here MPI_INT,
- Fourth argument: the computing unit that sends the message.
- Fifth argument: a mark characterizing the message: the value 10 can be changed but it must keep the same for the reception.
- Sixth argument: the MPI communicator COMM_WORLD
To receive data between two units: `MPI_Recv` (& integer, 1, MPI_INT, 0,10, MPI_COMM_WORLD,& status);

First argument: address of the variable where the data will be stored (here the name is identical but it is not mandatory.

Second argument: number of elements: 1 by default, otherwise the number of elements of the array,

Third argument: the type of the variable, here MPI_INT,

Fourth argument: the computing unit that sends the message.

Fifth argument: a mark characterizing the message: the value 10 can be changed but it must keep the same for the reception.

Sixth argument: the MPI communicator COMM_WORLD

Seventh argument: an integer variable that is returned indicating whether the message is well received.
To send a message from one unit to all others, the instruction to use is the broadcast.

```c
#include<stdio.h>
#include<mpi.h>
char processor_name[MPI_MAX_PROCESSOR_NAME];
int numprocs,myid,namelen;
MPI_Status status;

int main (int argc, char *argv[]) {

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    double reel=(double) myid;
    printf("Hello, Process %d of %d on %s valeur de reel %e\n", myid, numprocs, processor_name, reel);
    MPI_Bcast(&reel,1, MPI_DOUBLE,3, MPI_COMM_WORLD);
    MPI_Barrier( MPI_COMM_WORLD);
    printf("Hello, Process %d of %d on %s valeur de reel %e\n", myid, numprocs, processor_name, reel);
    MPI_Finalize();
}
```
The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);
The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);

- First argument: address of the variable for the value is stored
The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);

- First argument: address of the variable for the value is stored
- Second argument: size of the variable type
The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);

First argument: address of the variable for the value is stored
Second argument: size of the variable type
Third argument: Type of variable; here MPI_DOUBLE
Message sending: broadcast(2)

- The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);
- First argument: address of the variable for the value is stored
- Second argument: size of the variable type
- Third argument: Type of variable; here MPI_DOUBLE
- Fourth argument: number of the unit sending the global message
The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);

- First argument: address of the variable for the value is stored
- Second argument: size of the variable type
- Third argument: Type of variable; here MPI_DOUBLE
- Fourth argument: number of the unit sending the global message
- Fifth argument: communicator
Message sending: scatter
Message sending: gather

The diagram illustrates the process of MPI_GATHER(). It shows four processes, labeled P0, P1, P2, and P3, each holding a part of the data (A0, A1, A2, A3). The data is distributed among these processes, and the MPI_GATHER() function is used to gather this data into a single process. The arrows indicate the direction of data transfer, showing how the data is collected into a single location (Process 0 in this case).

The nodes labeled 0, 1, 2, and 3 represent the processes, with arrows indicating the communication paths. The labels A0, A1, A2, A3 on the arrows show the data being transferred.
Message sending: allgather
Message sending: gatherv

![Diagram showing message sending with MPI_gatherv function]
Reduction

- Operations can be performed at the time of collecting information on other calculation units.
Reduction

- Operations can be performed at the time of collecting information on other calculation units.
- The list of predefined operations are: sum, product, search for maximum or minimum and some other operations.

To do this, the statement is MPI Reduce with a list of arguments.
To do a reduction with an overall propagation of all the results, we can use MPI Reduce followed by MPI Bcast or more simply MPI AllReduce.
Operations can be performed at the time of collecting information on other calculation units

The list of predefined operations are: sum, product, search for maximum or minimum and some other operations

To do this, the statement is MPI_Reduce with a list of arguments
Operations can be performed at the time of collecting information on other calculation units.

The list of predefined operations are: sum, product, search for maximum or minimum and some other operations.

To do this, the statement is MPI_Reduce with a list of arguments.

To do a reduction with an overall propagation of all the results, we can use MPI_Reduce followed by MPI_Bcast or more simply MPI_AllReduce.
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double f( double );

double f( double a ) { return (4.0 / (1.0 + a*a)); }

int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i,namealen;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum=0.0, x;
    double startwtime = 0.0, endwtime;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s\n",myid, processor_name);
    n = 1000000; h = 1.0 / (double) n;
    if (myid == 0) {startwtime = MPI_Wtime();}
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    for (i = myid + 1; i <= n; i += numprocs){
        x = h * ((double)i - 0.5);sum += f(x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0){
        printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        endwtime = MPI_Wtime();
        printf("wall clock time = %f\n",endwtime-startwtime);
    }
    MPI_Finalize();
    exit(0);}

Pascal Viot
Tutorial: parallel coding MPI
September 12, 2018 23 / 24
The MPI library has been developed for several Fortran, C, C++, and Python languages and many network hardware. It lacks debugging tools.

Some sites to consult:
http://www.open-mpi.org/
http://mpi-forum.org/
The MPI library has been developed for several Fortran, C, C++, Python languages and many network hardware. It lacks debugging tools.

Developing a parallel program is a much more delicate task than a typical sequential program.
Conclusion and References

- The MPI library has been developed for several Fortran, C, C++, Python languages and many network hardware. It lacks debugging tools.

- Developing a parallel program is a much more delicate task than a typical sequential program.

- These few rules can help manage the flow of information from simulated program execution.
The MPI library has been developed for several Fortran, C, C++, Python languages and many network hardware. It lacks debugging tools.

Developing a parallel program is a much more delicate task than a typical sequential program.

These few rules can help manage the flow of information from simulated program execution.

Some sites to consult: