# Tutorial: parallel coding MPI

Pascal Viot

October 22, 2020

# Generalities

- The single power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 4GHz.

# Generalities

- The single power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 4GHz.
- The recent evolution of the processors goes in the direction of several cores (several computing units inside a single chip)

# Generalities

- The single power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 4GHz.
- The recent evolution of the processors goes in the direction of several cores (several computing units inside a single chip)
- Scientific programs are sometimes calculations which can be performed independently for some parts and therefore can use this available power.

# Generalities

- The single power of a processor is still growing, but at a slower rate because the frequency maximum processors is still limited to 4GHz.
- The recent evolution of the processors goes in the direction of several cores (several computing units inside a single chip)
- Scientific programs are sometimes calculations which can be performed independently for some parts and therefore can use this available power.
- There are two main methods for making a program usable by many compute units: MPI and OPENMP

# Generalities (2)

- Use a library that will manage the exchange of information (and data) between computing units and thus draws a program according to the distribution of the calculations on available units and data exchanges between them. The reference library is MPI (Message Passing Interface).

# Generalities (2)

- Use a library that will manage the exchange of information (and data) between computing units and thus draws a program according to the distribution of the calculations on available units and data exchanges 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.

# Generalities (2)

- Use a library that will manage the exchange of information (and data) between computing units and thus draws a program according to the distribution of the calculations on available units and data exchanges 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 execution end

# Outline

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

# Outline

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

# Outline

- MPI: definition and different versions
- Definition of the world of communicator, initialization and execution end
- 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 execution end
- 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 execution end
- 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 version 1 and 2: backward compatibility. Version 3.1: June 21, 2015. Version 4.0 (2019)

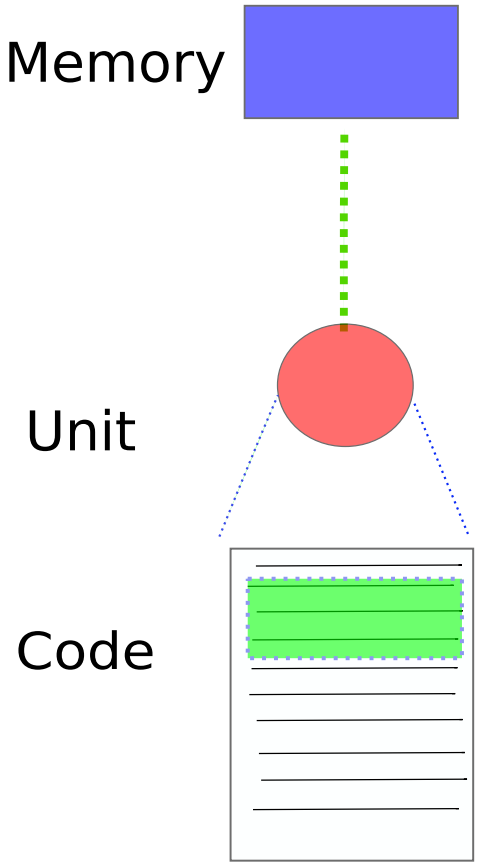
# MPI: definition et different versions

- MPI version 1 and 2: backward compatibility. Version 3.1: June 21, 2015. Version 4.0 (2019)
- Site: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Doc version 3.1, a downloadable document of 868 pages.

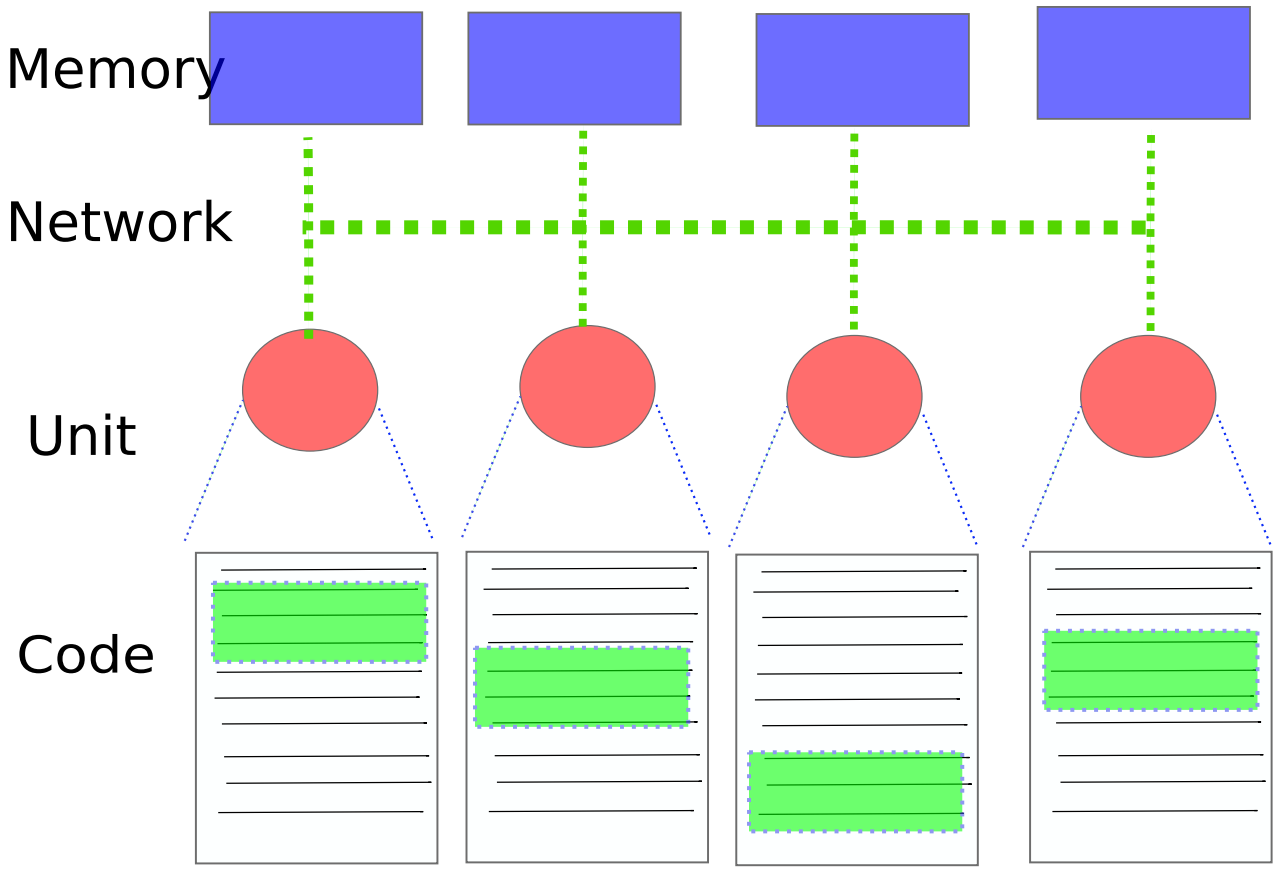
# MPI: definition et different versions

- MPI version 1 and 2: backward compatibility. Version 3.1: June 21, 2015. Version 4.0 (2019)
- Site: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Doc version 3.1, a downloadable document of 868 pages.
- OPENMPI contains many physical media. It is in the form of a compiled package for the GNU compiler. Version 4.0.3 on an Ubuntu version 20.04

# Communicator: Definition, initialization and execution e



# Definition of the communicator, initialization and execution end(2)



# Definition of the communicator, initialization and execution end(3)

```
#include<iostream>
#include<mpi.h>
int main (int argc, char *argv[])
{
int numprocs,myid,namelen;
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);
std::cout<<"Hello, Process"<<myid<<" of "<<numprocs<<"
    on "<< processor_name<<std::endl;
MPI_Barrier( MPI_COMM_WORLD);
MPI_Finalize();
exit(0);
}
```

# Definition of the communicator, initialization and execution end (4)

- the MPI_Init function initializes the units of the comunicator

# Definition of the communicator, initialization and execution end (4)

- the MPI_Init function initializes the units of the comunicator
- The MPI_Finalize function destroys the communicator

# Definition of the communicator, initialization and execution end (4)

- the MPI_Init function initializes the units of the comunicator
- The MPI_Finalize function destroys the communicator
- the MPI_Barrier function is a signal that allows all compute units to be synchronized

# Definition of the communicator, initialization and execution end (4)

- the MPI_Init function initializes the units of the comunicator
- The MPI_Finalize function destroys the communicator
- 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

# Definition of the communicator, initialization and execution end (4)

- the MPI_Init function initializes the units of the comunicator
- The MPI_Finalize function destroys the communicator
- 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 execution end (4)

- the MPI_Init function initializes the units of the comunicator
- The MPI_Finalize function destroys the communicator
- 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.cc -O3 -o hello

  (or mpic++)

# Compiling and running a parallel code

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

  (or mpic++)
- For running a parallel code, write in a terminal window
  mpirun -np 8 hello

# Compiling and running a parallel code

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

  (or mpic++)
- For running a parallel code, write in a terminal window
  mpirun -np 8 hello
- If the number of virtual codes exceeds the number of physical cores
  mpirun -oversubscribe -n 8 hello

# Compiling and running a parallel code

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

  (or mpic++)
- For running a parallel code, write in a terminal window
  mpirun -np 8 hello
- If the number of virtual codes exceeds the number of physical cores
  mpirun -oversubscribe -n 8 hello
- In practice, it is better not to exceed a number of units higher than the number of hearts of the machine

# Physical limitations of a parallel program

- To exchange information between computing units, the library has specific instructions to send messages.

# Physical limitations of a parallel program

- 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 by the network capacity (e.g. 1Gb/s) and is much lower than what happens for cpu-memory exchanges.

# Physical limitations of a parallel program

- 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 by the network capacity (e.g. 1Gb/s) and is much lower than what happens for cpu-memory exchanges.
- You can send messages either between two calculation units (point-to-point message) or globally (eg the first unit sends to all others or all send information to all. In the latter case, we can observe significant performance falls.

# Message sending: point-to-point communication

```
#include<iostream>
#include<mpi.h>
int main (int argc, char *argv[])
{
  int numprocs,myid,namelen;
  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);
  MPI_Status status;
  int entier=myid;
  std::cout<<"Before " <<myid<<" of "<<numprocs<<
    " on "<< processor_name<<" integer value "<<entier<<std::endl;
  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);
  }
  std::cout<<"After  " <<myid<<" of "<<numprocs<<
    " on "<< processor_name<<" integer value "<<entier<<std::endl;
  MPI_Barrier( MPI_COMM_WORLD);
  MPI_Finalize();
}
```

# Message sending: point-to-point communication (2)

- To send data between two units: MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);

# Message sending: point-to-point communication (2)

- To send data between two units: MPI_Send(& integer, 1, MPI_INT, 3,10, MPI_COMM_WORLD);
- First argument: address of the variable

# Message sending: point-to-point communication (2)

- 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,

# Message sending: point-to-point communication (2)

- 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,

# Message sending: point-to-point communication (2)

- 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.

# Message sending: point-to-point communication (2)

- 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.

# Message sending: point-to-point communication (2)

- 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);

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

# 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);
- 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,

# 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);
- 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,

# 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);
- 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.

# 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);
- 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.

# 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);
- 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

# 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);
- 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.

# Message sending: broadcast

To send a message from one unit to all others, the instruction to use is the broadcast.

```
#include<iostream>
#include<mpi.h>
int main (int argc, char *argv[])
{
  int numprocs,myid,namelen;
  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);
  double reel=(double) myid;
  std::cout<<"Before " <<myid<<" of "<<numprocs<<
  " on "<< processor_name<<" integer value "<<reel<<std::endl;
  MPI_Bcast(&reel,1, MPI_DOUBLE,3, MPI_COMM_WORLD);
  MPI_Barrier( MPI_COMM_WORLD);
  std::cout<<"After  " <<myid<<" of "<<numprocs<<
  " on "<< processor_name<<" integer value "<<reel<<std::endl;
  MPI_Finalize();
}
```

# Message sending: broadcast(2)

- The different arguments of the statement: MPI_Bcast (& reel, 1, MPI_DOUBLE, 3, MPI_COMM_WORLD);

# 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

# 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

# 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

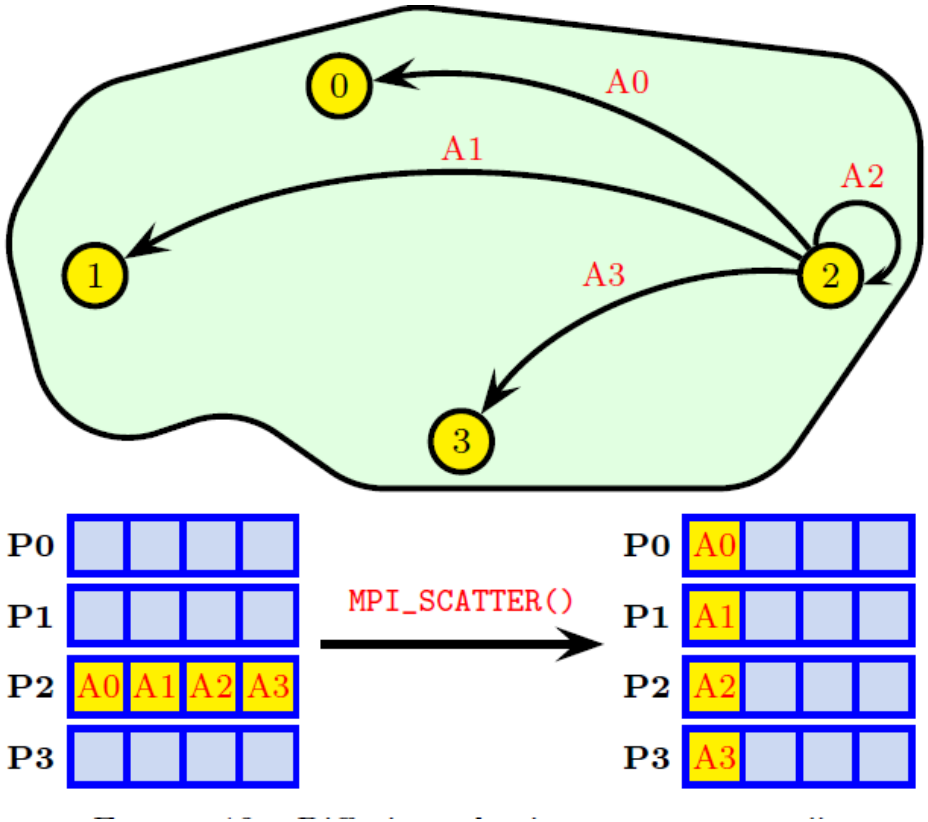
# Message sending: broadcast(2)

- 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

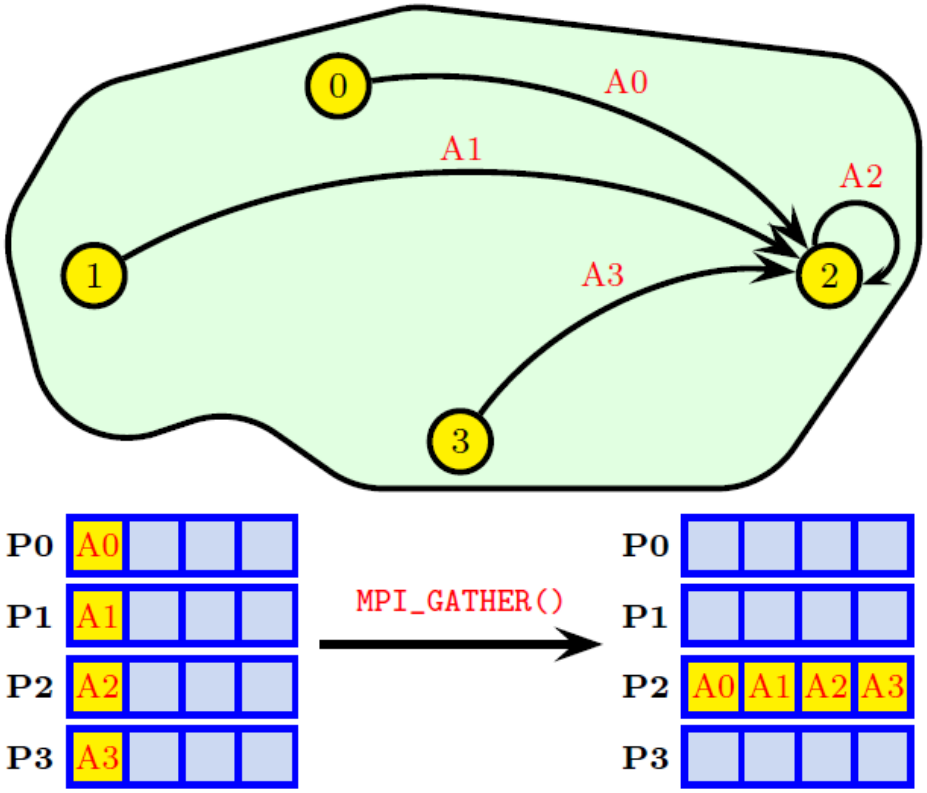
# Message sending: broadcast(2)

- 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
- Fifth argument: communicator

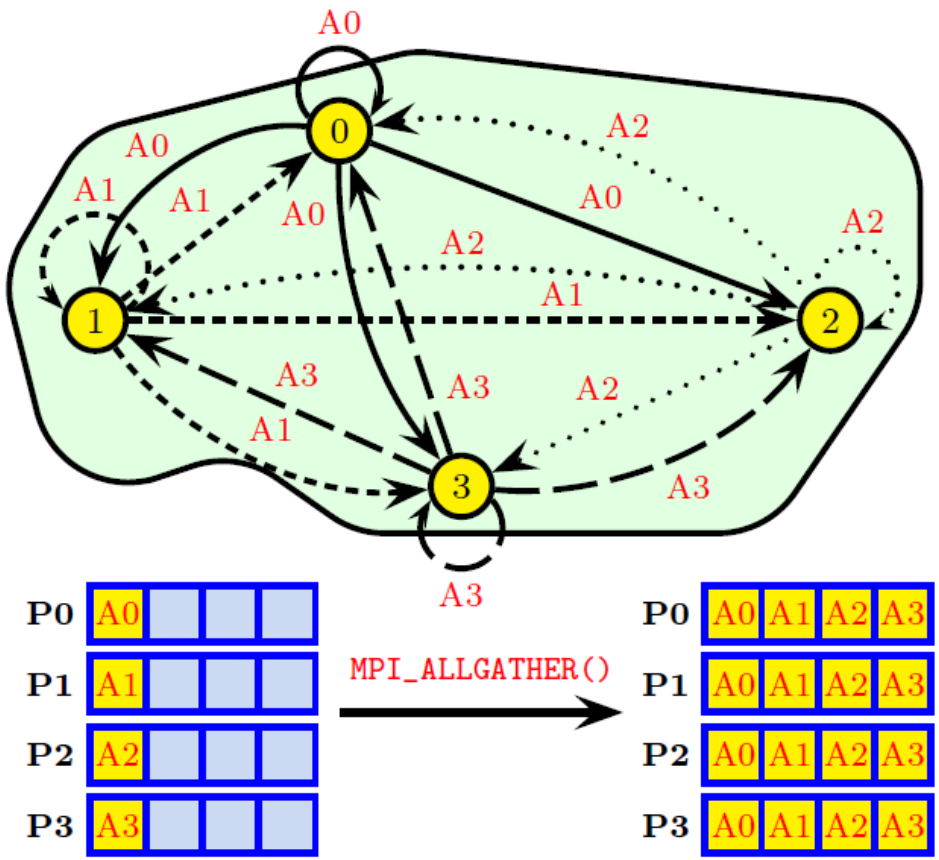
# Message sending: scatter



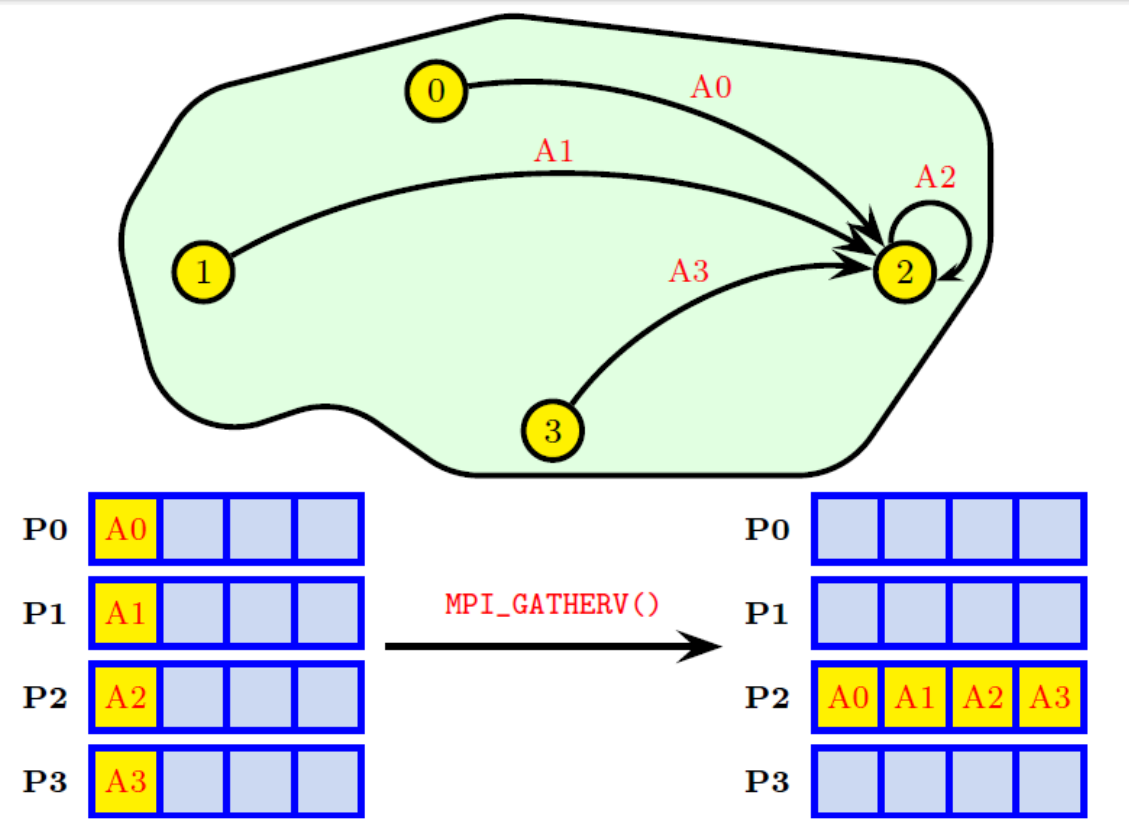
# Message sending: gather



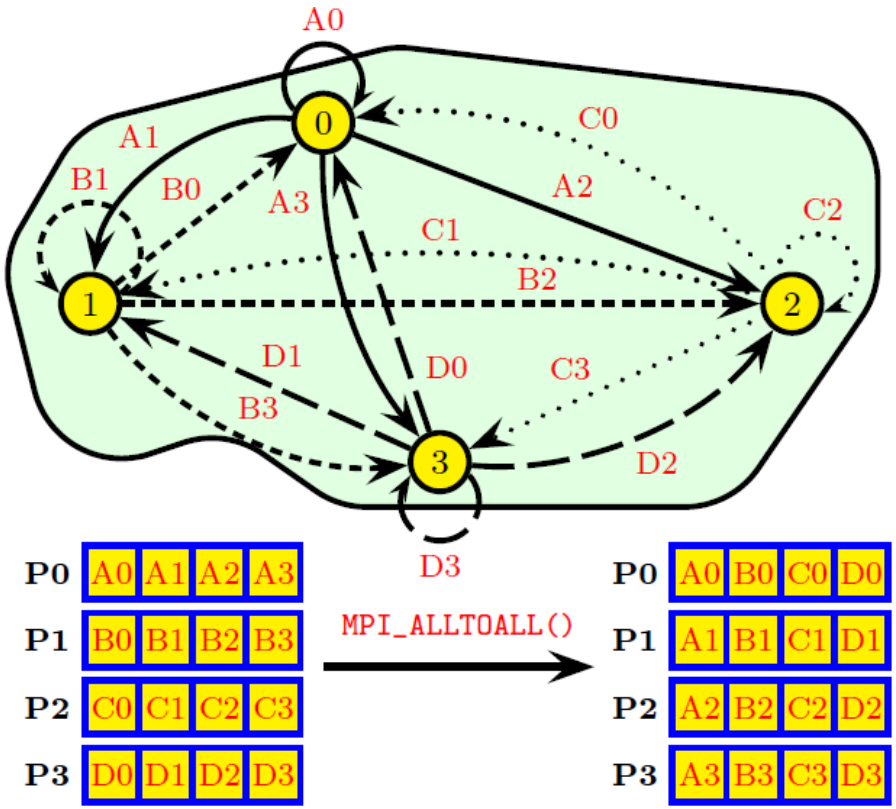
# Message sending: allgather



# Message sending: gatherv



# Message sending: alltoall



# Reduction

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

# Reduction

- 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

# Reduction

- 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

# First scientific program

```
#include<iostream>
#include <iomanip>
#include<cmath>
#include<mpi.h>
double f( double a ) { return (4.0 / (1.0 + a*a)); }
int main (int argc, char *argv[])
{
int myid, numprocs,namelen;
double pi, sum=0.0;
double startwtime = 0.0;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
int n = 1000000000;
double h   = 1.0 / (double) n;
if (myid == 0)     {startwtime = MPI_Wtime();}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (int i = myid + 1; i <= n; i += numprocs){
  double x = h * (i - 0.5);
  sum += f(x);}
double mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0){
  std::cout<<"pi is approximately equal "<<std::setprecision(16) << pi <<"  Error is "<<
       fabs(pi - M_PI)<<std::endl;
 std::cout<<"Wall clock time = "<<MPI_Wtime()-startwtime<<std::endl;
}
MPI_Finalize();
exit(0);}
```

# Conclusion and References

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

# 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.

# 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.

# 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.
- Some sites to consult: http://www.open-mpi.org/ http://mpi-forum.org/