CARLA 2021
LATIN AMERICA HIGH PERFORMANCE COMPUTING CONFERENCE
VIRTUAL EDITION

COMITTEE BOARD

Beginners MPI Tutorial

Gilberto Díaz
Universidad Industrial de Santander
Robinson Rivas Suárez
Universidad Central de Venezuela
Esteban Mocskos
Universidad de Buenos Aires

CARLA 2021
LATIN AMERICA HIGH PERFORMANCE COMPUTING CONFERENCE
VIRTUAL EDITION

Message Passing Interface

Based on material from: Gilberto Díaz

gilberto.diaz@uis.edu.co

Universidad Industrial de Santander
Super Computación y Cálculo Científico (SC3)
http://www.sc3.uis.edu.co
Bucaramanga - Colombia

Gilberto

System Engineer (Mérida, Venezuela)

Master in Computer Science (Mérida, Venezuela)

PhD Candidate in Computer Science (Bucaramanga, Colombia)

ICT manager and advisor, Universidad Industrial de Santander

Esteban

BsC in Computer Science (Buenos Aires, Argentina)

Master in Computer Science (Buenos Aires, Argentina)

PhD in Computer Science (Buenos Aires, Argentina)

Professor and Researcher, Universidad de Buenos Aires

Robinson

BsC in Computer Science (Maracaibo, Venezuela)

Master in Computer Science (Caracas, Venezuela)

Network Engineer (OIC, Japan)

Director, Computer Science School, Universidad Central de Venezuela

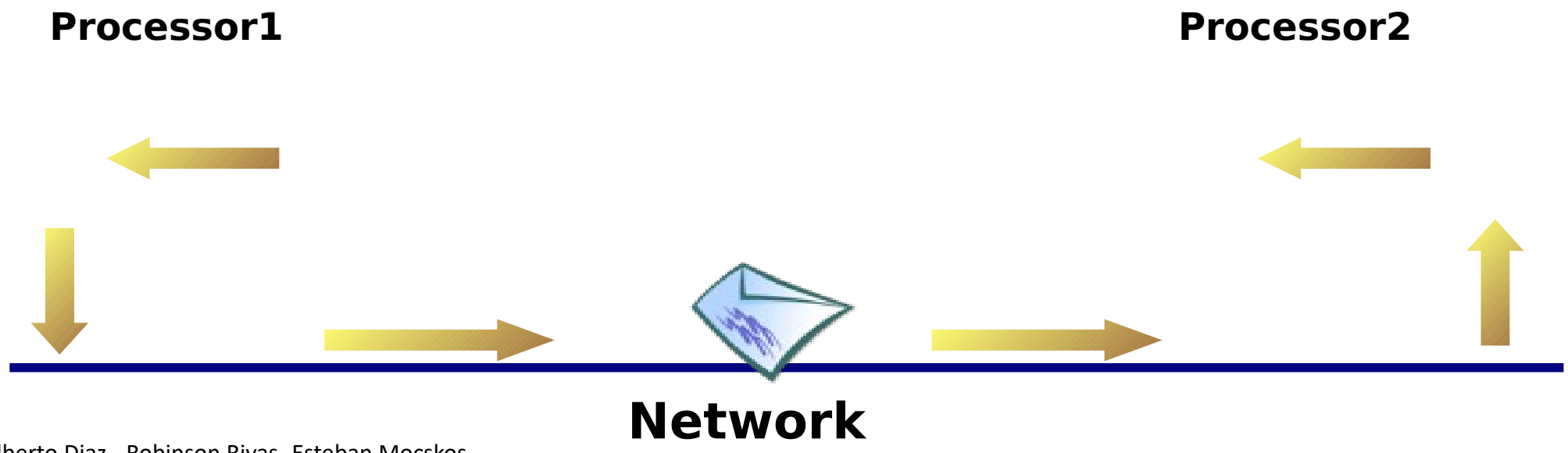
Message Passing Interface

MPI is an standard for message passing systems.

It is designed by a group of researchers from academic and private companies to create portable code capable to run on different parallel platforms.

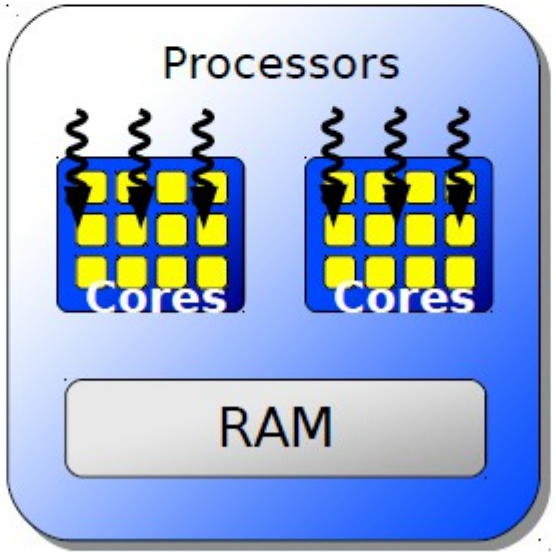
Message Passing Interface

MPI was designed to work on distributed memory architectures where every computer has its own RAM and the only way to exchage information is sending messages through the network.



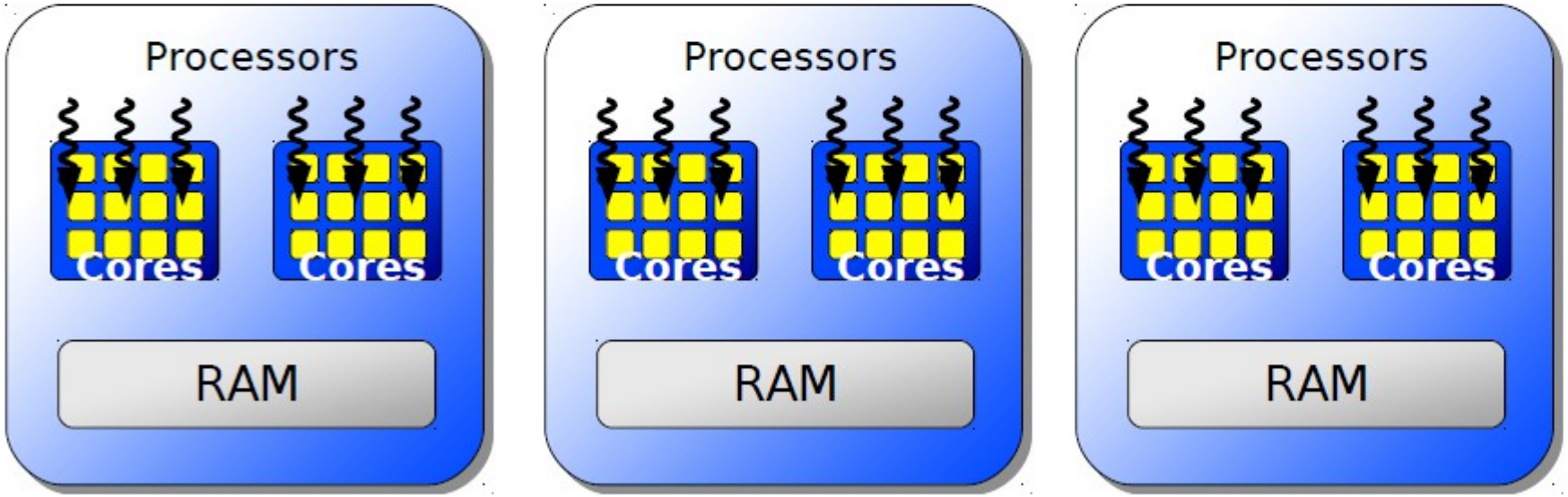
Message Passing Interface

- Some problems have a such dimension that local resources are not enough, no matters how plenty they are.



Message Passing Interface

- Therefore, we need to extend, somehow, our computing power using remote resources.



OpenMP vs MPI

OpenMP

- **Easy programming model**
- **Few modifications to original code**
- **Short development time**
- **Scale only to local resources**

MPI

- **We have to rewrite our program**
- **Difficult programming model**
- **Longer development time**
- **Scale very well using distributed resources.**
- **We need to know very well the platform.**

Message Passing Interface

However, it is possible to run MPI programs on shared memory architectures.

This standard defines the syntax and the semantic for a set of rutines.

Message Passing Interface

Some popular implementations of the MPI standard we have:

- MPICH
- LAM
- OpenMPI
- MVAPICH

OpenMPI

OpenMPI is supported by a big group of organizations.

OpenMPI

Main objectives of OpenMPI design

- Provide portable code.
- Provide efficient implementations.
- Support heterogeneous parallel architectures.

How does MPI work?

- Data level parallelism

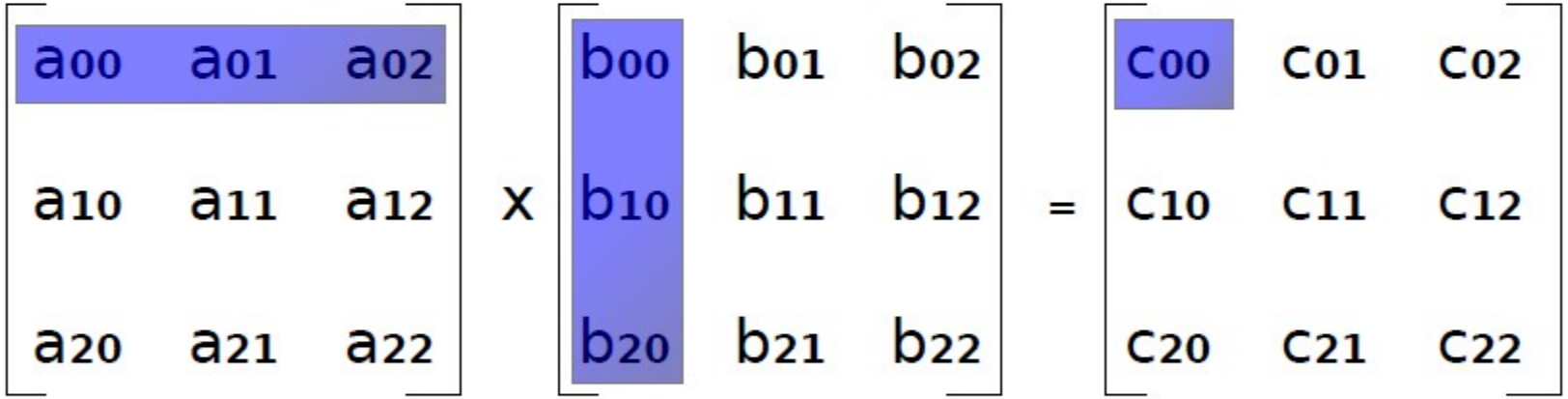
Suppose we have a matrixes multiplication

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix}$$

How does MPI work?

- Data level parallelism

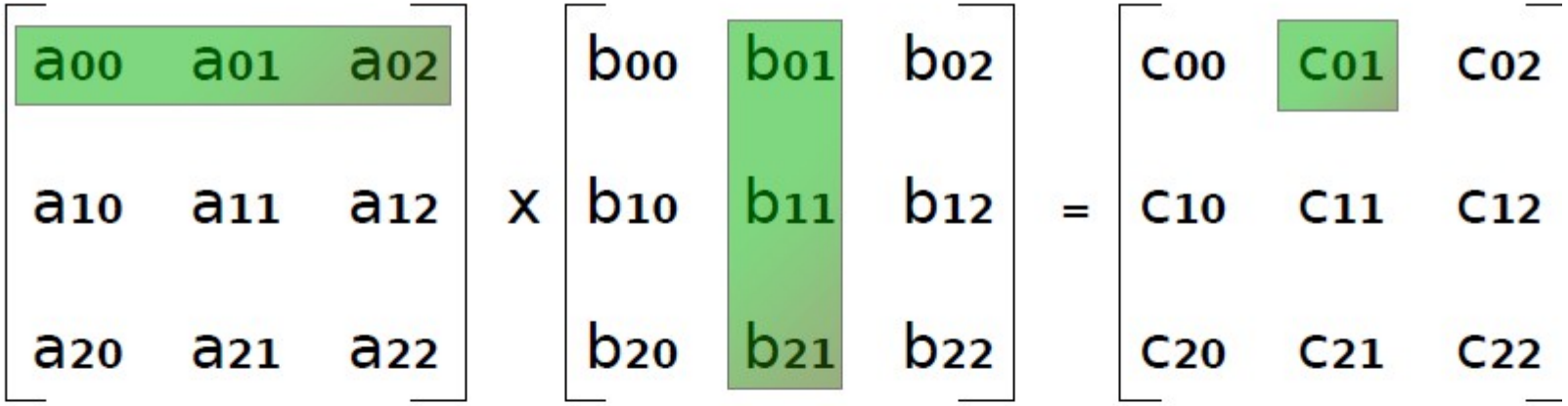
Suppose we have a matrixes multiplication



How does MPI work?

- Data level parallelism

Suppose we have a matrixes multiplication



How does MPI work?

- Data level parallelism

Suppose we have a matrixes multiplication

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix}$$

How does MPI work?

- Data level parallelism

Suppose we have a matrixes multiplication

$$c_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20}$$

$$c_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21}$$

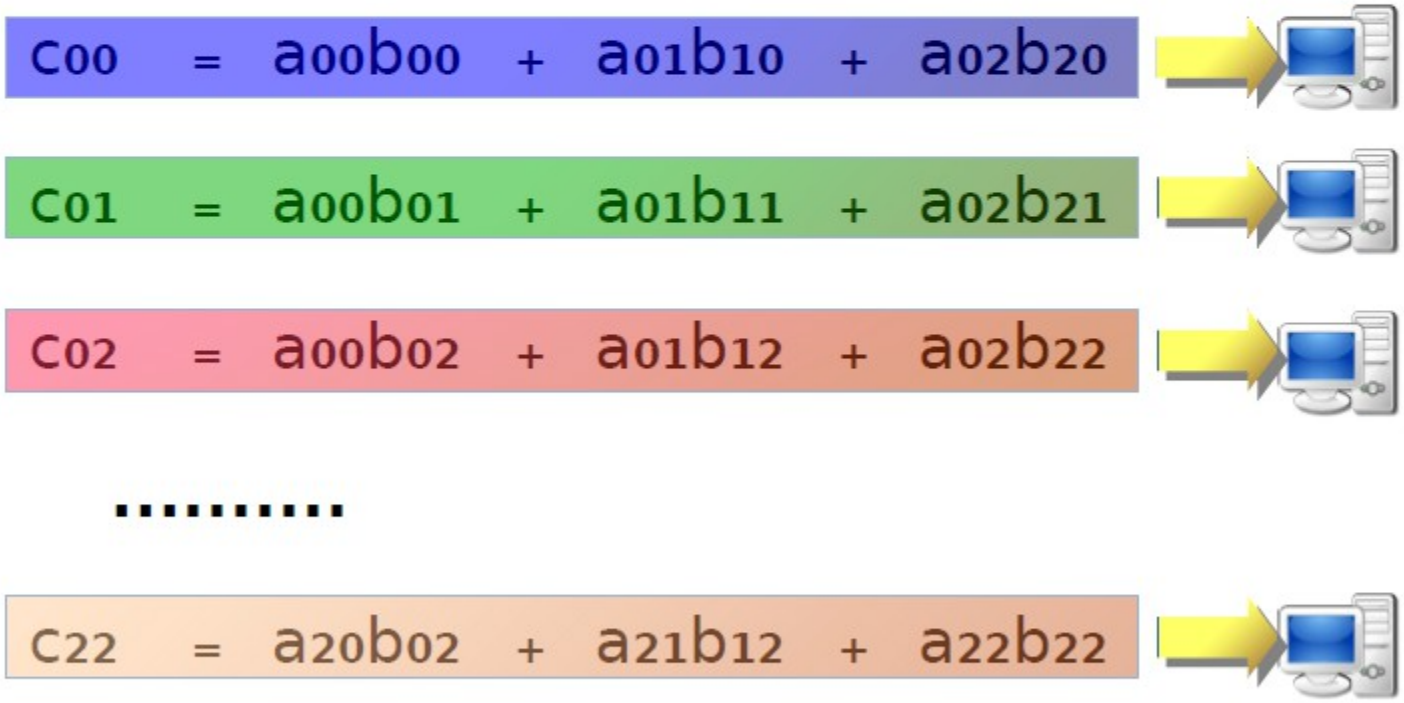
$$c_{02} = a_{00}b_{02} + a_{01}b_{12} + a_{02}b_{22}$$

$$\ldots\ldots\ldots$$

$$c_{22} = a_{20}b_{02} + a_{21}b_{12} + a_{22}b_{22}$$

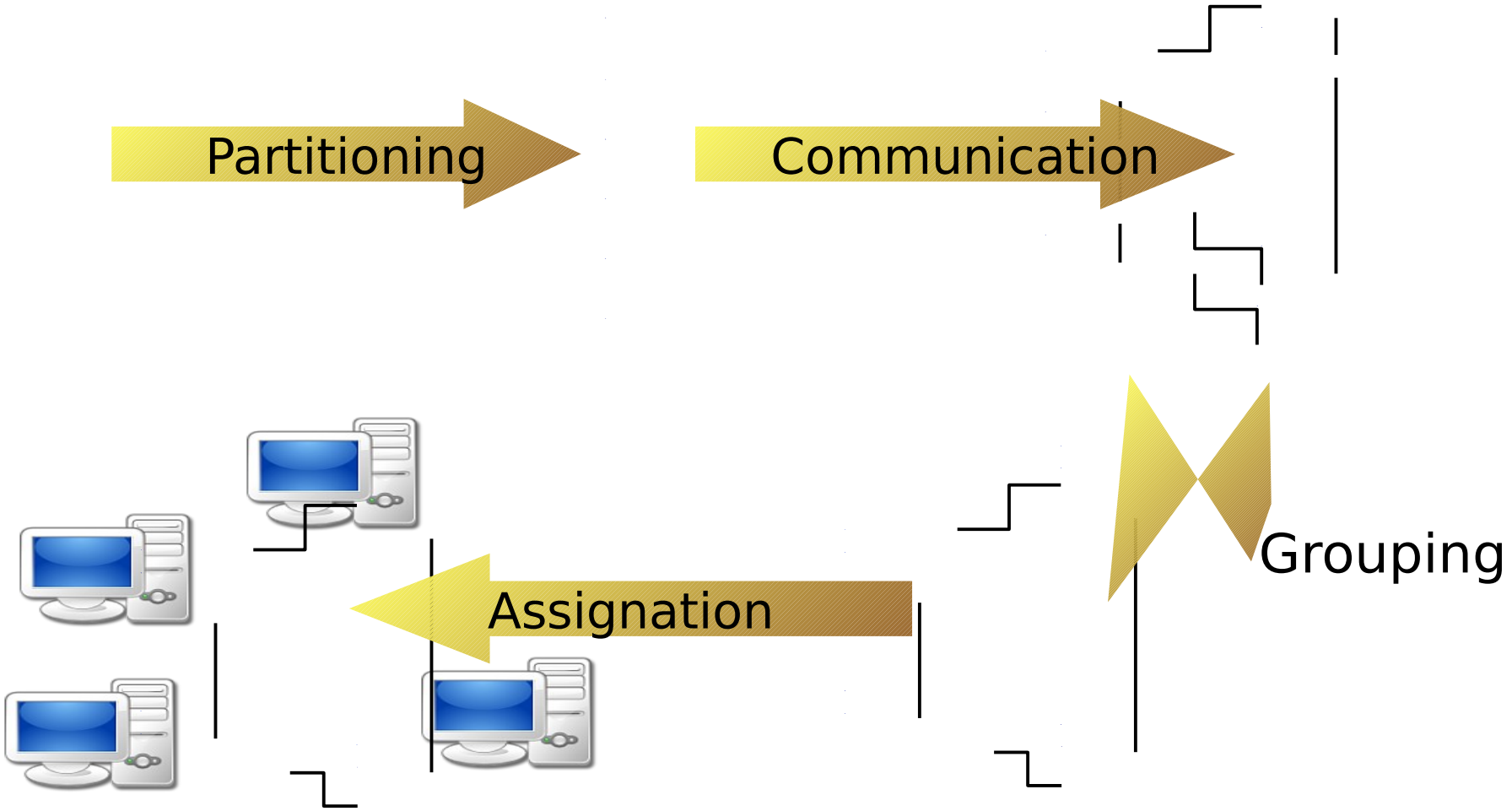
How does MPI work?

- Data level parallelism



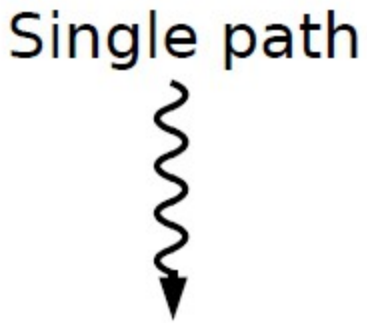
How does MPI work?

Problem



Serial Program

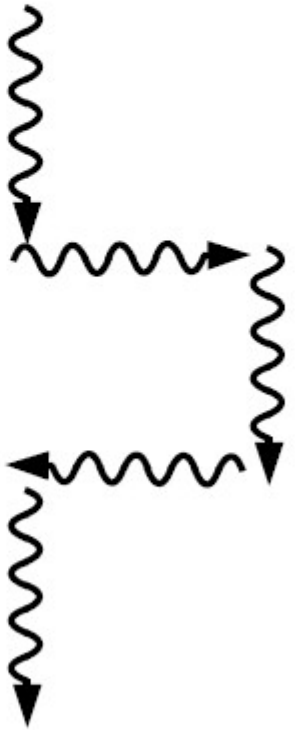
```
#include <stdio.h>

int main(int argc, char *argv[]){
      doSomeWork();
      return 0;
}
```



Serial Program

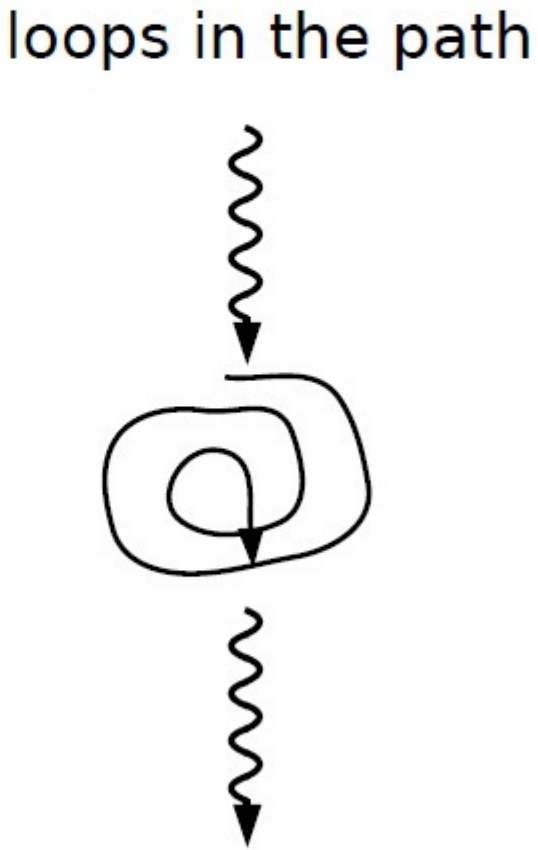
```
#include <stdio.h>

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

        i = atoi(argv[1]);
        if( i == 0 ){
                doSomeWork();
        }
        else{
                doSomeOtherWork();
        }
        return 0;
}
```

Curves in the path



Serial Program

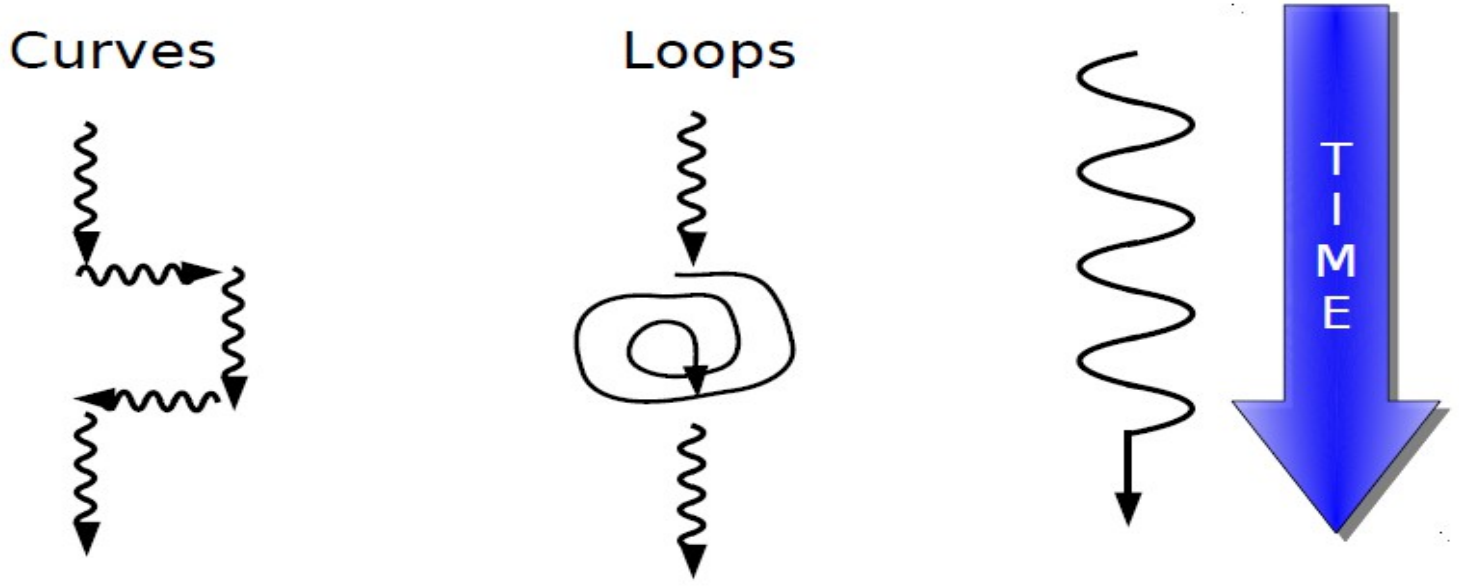
```
#include <stdio.h>

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

        i = atoi(argv[1]);
        while( i <= 10 ){
                doSomeWork();
        }
        return 0;
}
```



Serial Program

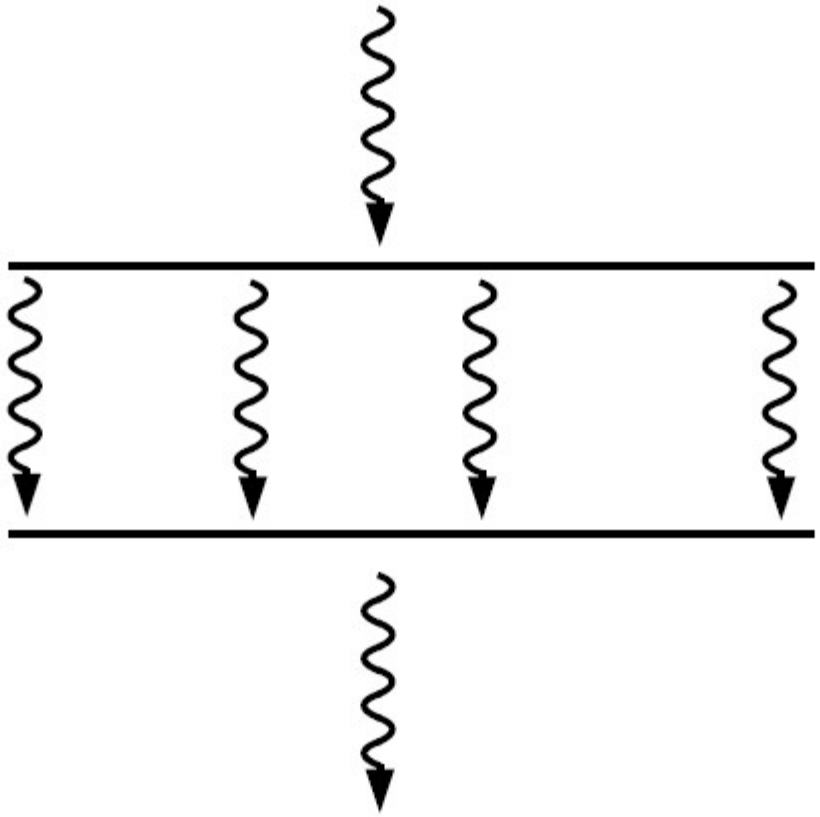
No matters the shape of the flow, one instruction

is executed after each other.



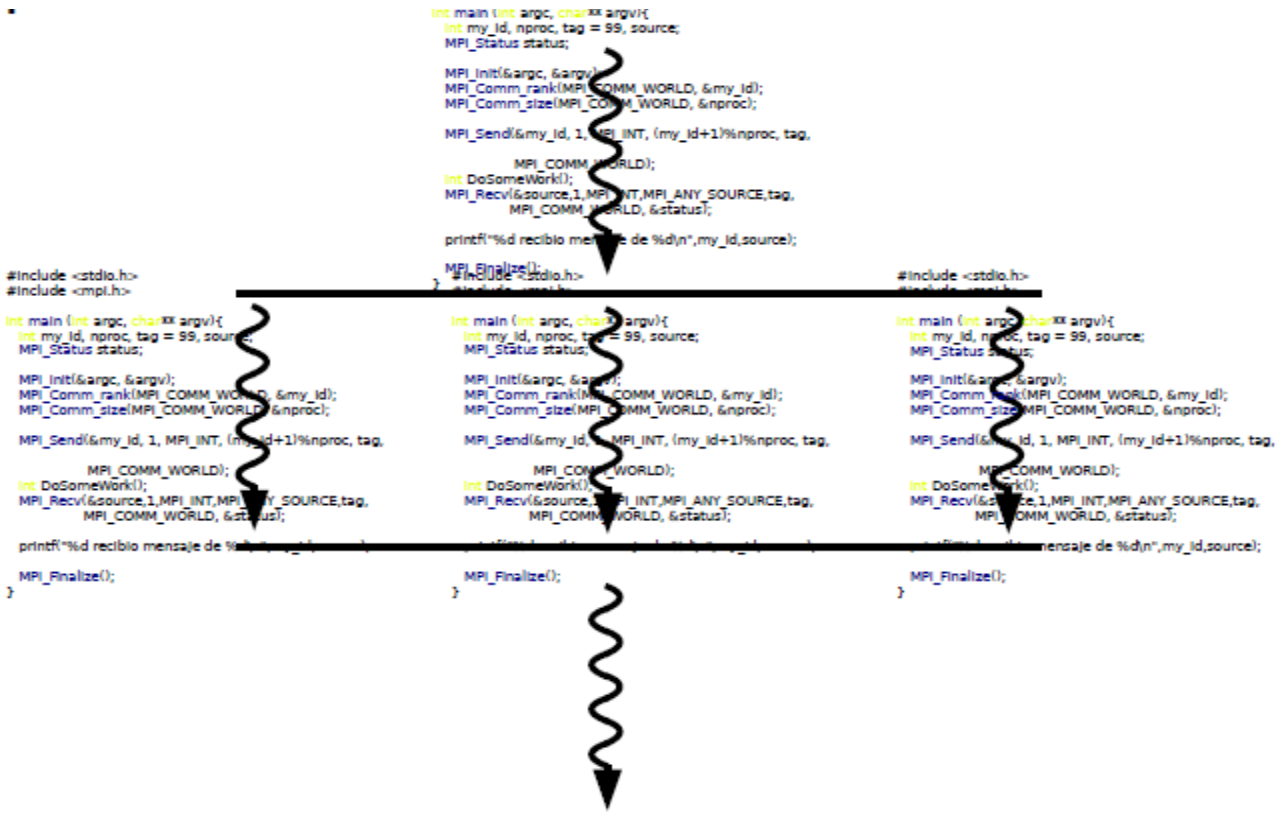
MPI Parallel Program

A parallel program has a different shape



MPI Parallel Program

There is only one code. That same code is executed by all processors



My first MPI program

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char** argv){
    int my_id, nproc, tag = 99, source;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    printf("I am processor %d of %d\n",
           my_id,nproc);
    MPI_Finalize();
}
```

How to compile a MPI program

The compilation procedure is different from the traditional way.
A wrapper is used to compile a MPI program

Examples:

```
mpicc program.c -o prog.exe
mpic++ program.cpp -o prog.exe
mpif77 program.f -o prog.exe
pif90 program.f90 -o prog.exe
```

How to run a MPI program

A MPI program execution is different than common programs.

mpirun –*machinefile* machines *-np* 4 program.exe

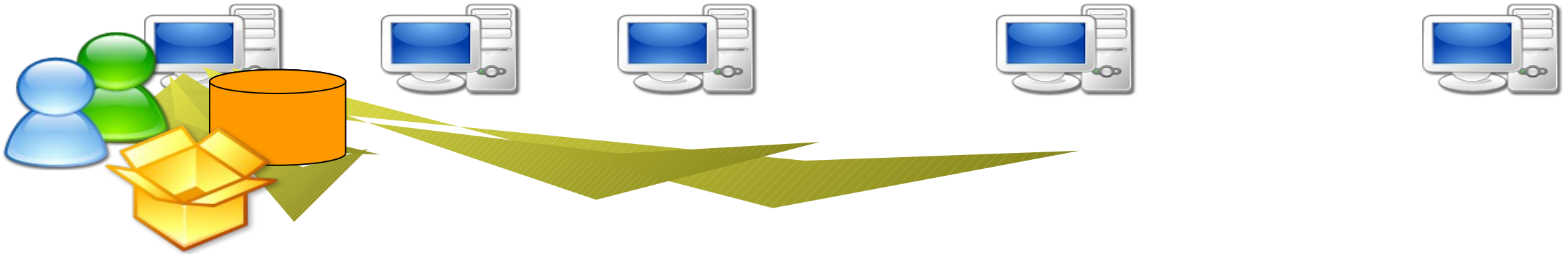
The execution procedure is independent from the implementation.
It is necessary to set some environment variables:

- PATH
- LD_LIBRARY_PATH

Requirements for execution

In order to execute a MPI program it is necessary set a proper environment:

- A common File System where all files are located: applications, user files, input and output data.
- A centralized system for user authentication.
- Possibility for remote command execution



MPI Program Example

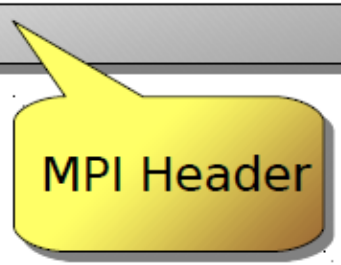
```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char** argv){
   int my_id, nproc, tag = 99, source;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &nproc);

   MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag, MPI_COMM_WORLD);
   int DoSomeWork();
   MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag, MPI_COMM_WORLD, &status);

   printf("%d receive message from %d\n",my_id,source);

   MPI_Finalize();
}
```

Example 0

- Compile and run mpi0.c
- Try different number of nodes (2,4,8)

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
           MPI_COMM_WORLD);
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
           MPI_COMM_WORLD, &status);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```



MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;
```

MPI Header

Header files contains:

- MPI data types definitions
- MPI functions prototypes definitions
  - **mpi.h** for C and C++
  - **mpif.h** for Fortran77 and Fortran90

```
}
```

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
           MPI_COMM_WORLD);
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
           MPI_COMM_WORLD, &status);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```

Parallel Section Initialization

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```

Parallel Section Initialization

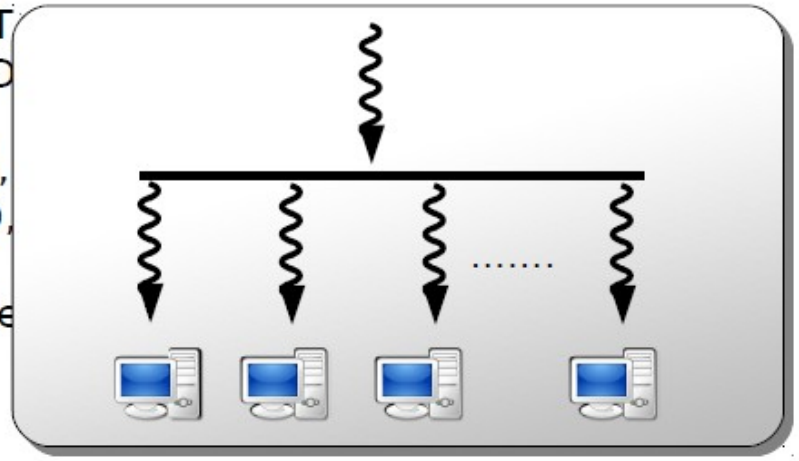
- Initialize the parallalel section of the program

  MPI_Init(&argc, &argv)

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT
           MPI_COMM_WORLD
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,
           MPI_COMM_WORLD,

  printf("%d recibio mensaje de

  MPI_Finalize();
}
```

Parallel Section Initialization

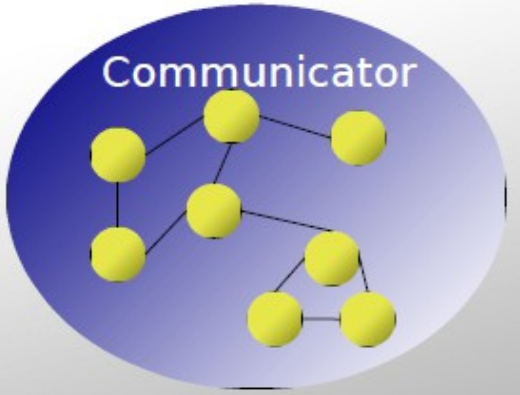


MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Parallel Section Initialization

MPI communicator is a variable to make reference to the whole group of processes created by the execution command

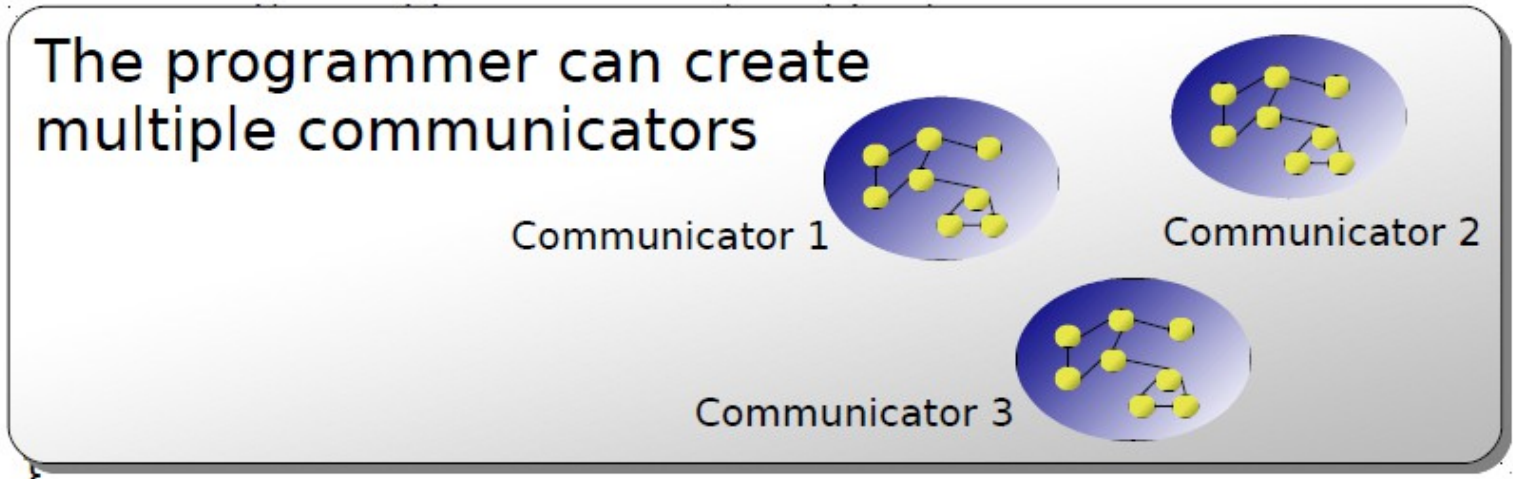


MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Parallel Section Initialization

The programmer can create multiple communicators

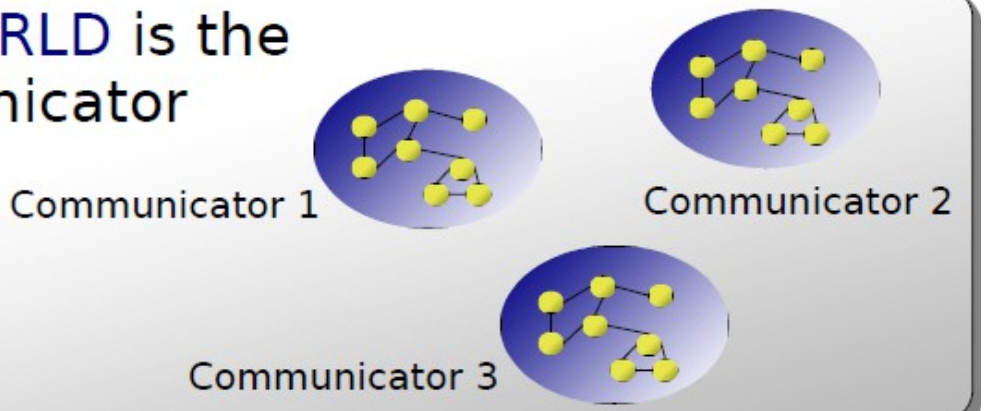


MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Parallel Section Initialization

MPI_COMM_WORLD is the default communicator

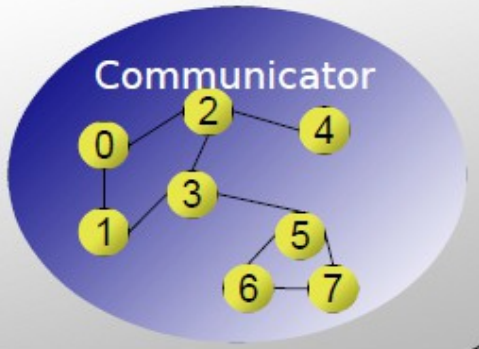


MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

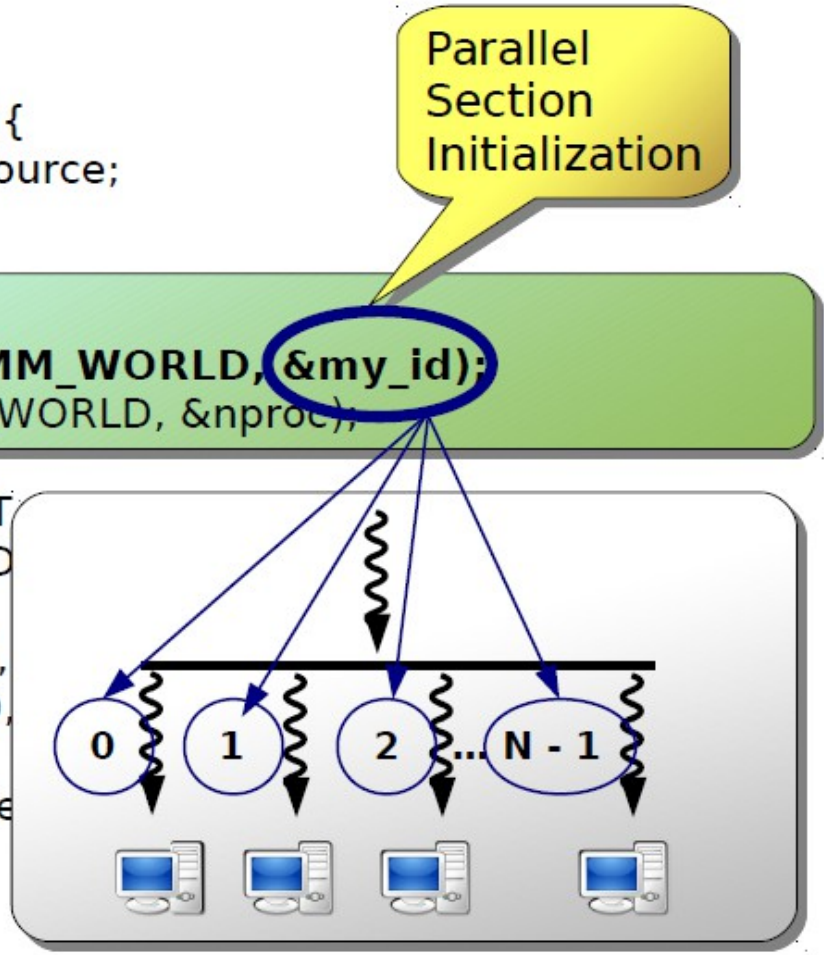
Parallel Section Initialization

MPI_Comm_rank(MPI_COMM_WORLD, &my_id)
Identify the local process



MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT
           MPI_COMM_WORLD
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,
           MPI_COMM_WORLD,

  printf("%d recibio mensaje de

  MPI_Finalize();
}
```

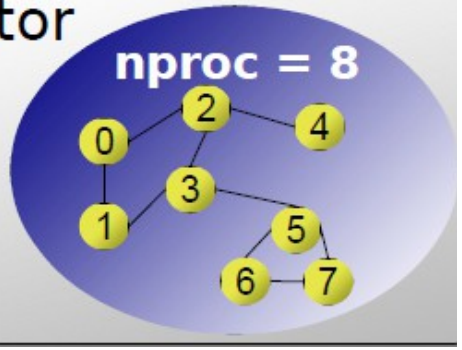


MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Parallel Section Initialization

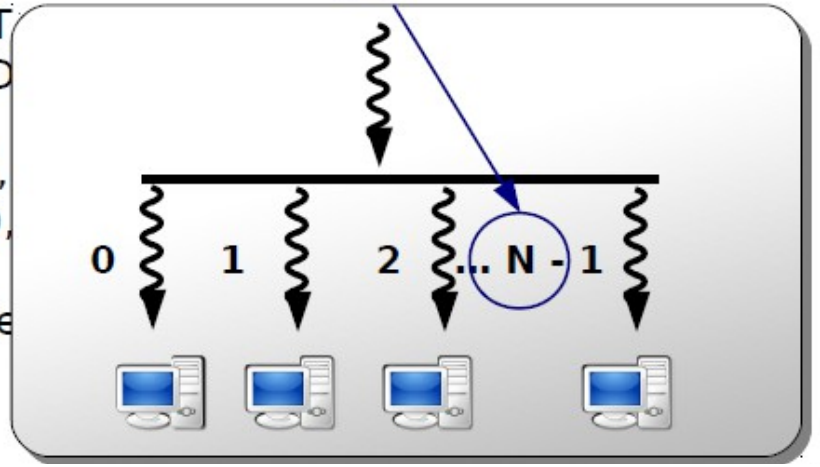
MPI_Comm_size(MPI_COMM_WORLD, &nproc)
Save the size of the communicator in *nproc* variable



MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT
           MPI_COMM_WORLD
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,
           MPI_COMM_WORLD,

  printf("%d recibio mensaje de

  MPI_Finalize();
}
```

Parallel Section Initialization



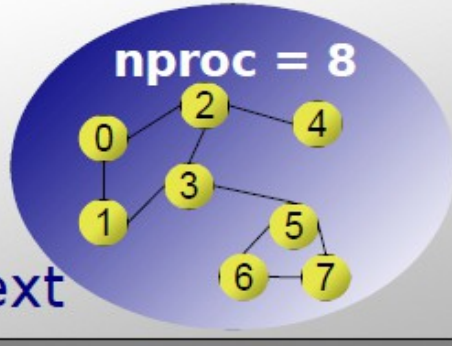
MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

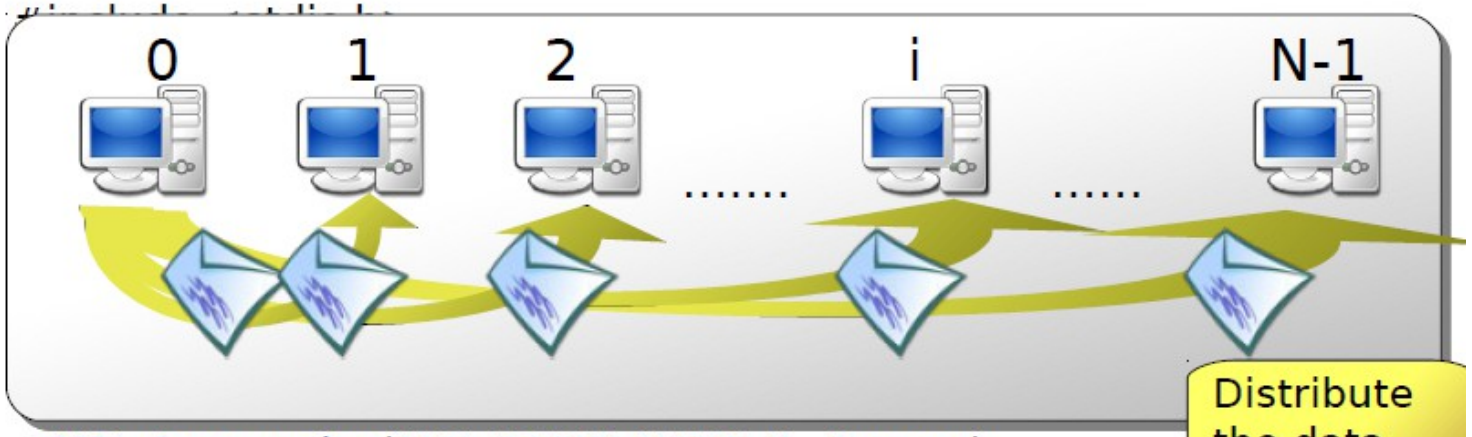
Parallel Section Initialization

- Processes can be collected into groups
- Each message is sent in a specific context

Communicator = Group + Context

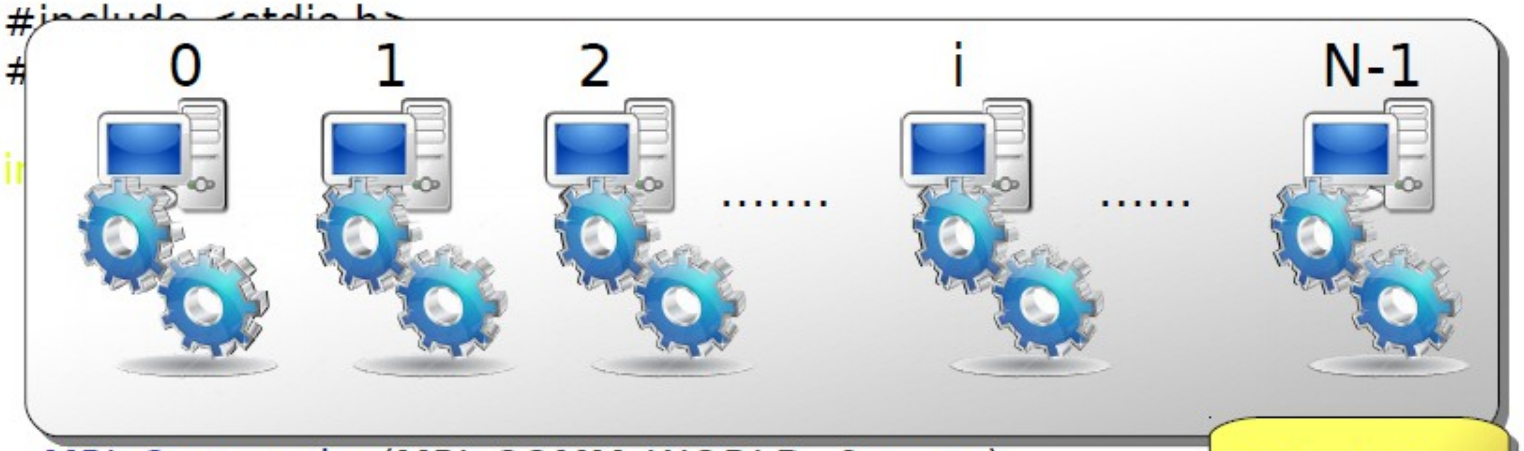


MPI Program Components




```
MPI_Comm_size(MPI_COMM_WORLD, &nproc);

MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
         MPI_COMM_WORLD);
int DoSomeWork();
MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
         MPI_COMM_WORLD, &status);

printf("%d recibio mensaje de %d\n",my_id,source);

MPI_Finalize();
}
```

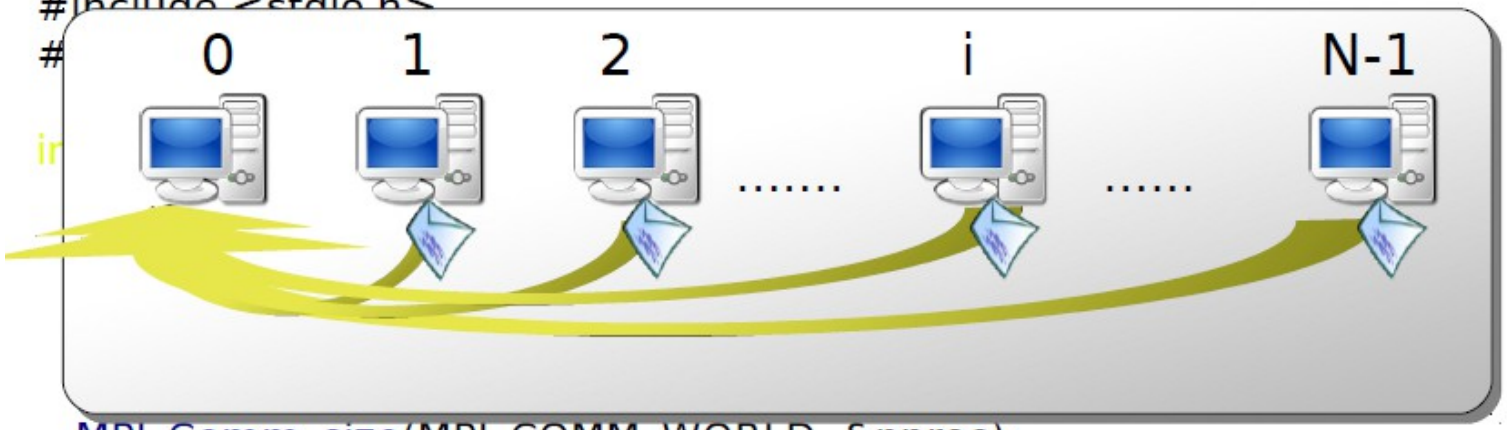
MPI Program Components




```
MPI_Comm_size(MPI_COMM_WORLD, &nproc);

MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
         MPI_COMM_WORLD);
int DoSomeWork();
MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
         MPI_COMM_WORLD, &status);

printf("%d recibio mensaje de %d\n",my_id,source);

MPI_Finalize();
}
```

Calculations

MPI Program Components




```
MPI_Comm_size(MPI_COMM_WORLD, &nproc);

MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
         MPI_COMM_WORLD);
int DoSomeWork();
MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
         MPI_COMM_WORLD, &status);

printf("%d recibio mensaje de %d\n",my_id,source);

MPI_Finalize();
}
```

Collect Partial Results

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag
           MPI_COMM_WORLD);
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
           MPI_COMM_WORLD, &status);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```

Print some Results

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
           MPI_COMM_WORLD);
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,ta
           MPI_COMM_WORLD, &status);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```

End of parallel Section

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_Status status;
```

- Ends the parallel section of the program

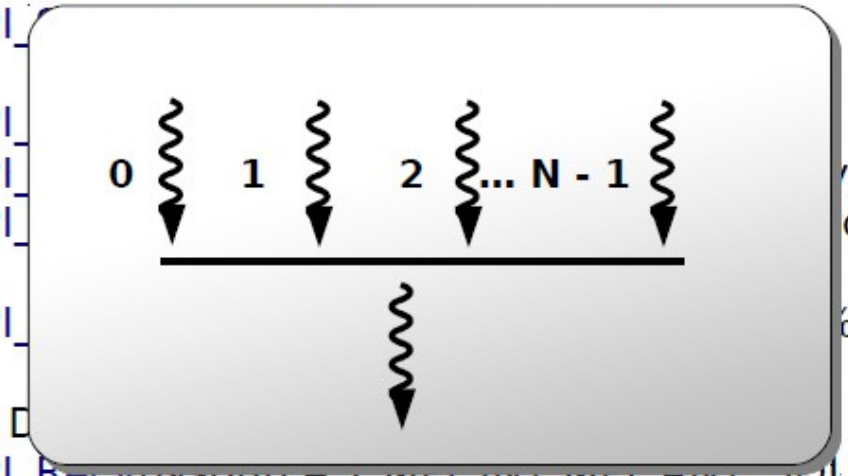
MPI_Finalize()

```
            MPI_COMM_WORLD);
  int DoSomeWork();
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,ta
            MPI_COMM_WORLD, &status);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```

End of parallel Section

MPI Program Components

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

int main (int argc, char** argv){
  int my_id, nproc, tag = 99, source;
  MPI_
  MPI_
  MPI_                                          _id);
  MPI_                                          oc);
  MPI_                                          nproc, tag,
  int D
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,ta
           MPI_COMM_WORLD, &status);

  printf("%d recibio mensaje de %d\n",my_id,source);

  MPI_Finalize();
}
```



End of parallel Section

Example 1

- Compile and run mpi1.c

```
MPI_Comm_rank (MPI_COMM_WORLD, &MyId);
MPI_Comm_size (MPI_COMM_WORLD, &N);
srand(MyId);
MPI_Get_processor_name(machine_name,&namelength);
value= (rand() % 100);
printf("Hi, I am process %d from %d. I calculate %d. \\
       Running on %s\n",MyId,N,value,machine_name);
```

- Discuss this results

MPI Programming Model

MPI implementations are libraries. All MPI instructions are functions calls

- Management functions: init, end, etc.
- Point to Point communication functions.
- Collective communications
- Data types management.

Installing OpenMPI

Installation procedure from source code is a simple task. Just follow these instructions:

Download the source code

cd /usr/local/src/

wget https://www.open-mpi.org/software/ompi/v4.0/downloads/openmpi-4.0.0.tar.bz2

Unpack the file

tar xvjf openmpi-.0.0.tar.bz2

Installing OpenMPI

Configure it

```
export CFLAGS="-I/usr/local/slurm/include/slurm -L/usr/local/slurm/lib"
./configure --prefix=/usr/local/openmpi-3.0.0/ --enable-static \
    --enable-mpi-fortran --with-slurm –with-pmi=/usr/local/slurm/ \
    --with-pmi-libdir=/usr/local/slurm/ --with-cuda  -enable-mpi-cxx \
    CC=icc CXX=icpc FC=ifort
```

Compile the sources

```
make
```

Finally, install the software

```
make install
```

Set environment variables:

PATH, LD_LIBRARY_PATH

Send and Receive Messages

The functions to send and receive manage the data using the first three arguments:

Memory address where the data is.

Size of the memory

Data type

MPI_Send(&buffer, size, MPI_INT,.....

MPI_Recv(&buffer, size, MPI_INT,.....

Send and Receive Messages

In MPI Messages can have an optional label (TAG) that helps programmer to identify groups of messages

For instance, you can group messages by function, for control, for requests, etc. If you don’t care about labels, use the default tag MPI_ANY_TAG

MPI Data types

There are several native data types:

MPI_BYTE
MPI_INT
MPI_CHAR
MPI_SHORT
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_UNSIGNED.....

MPI implementations provides functions to create new data types.

MPI_Send

Function to send a message

MPI_Send(*&a,n,MPI_INT,dest,tag,COMM*)

Where:

a Data to send
n Number of elements in *a*
MPI_INT Integer data type
dest ID of the destination process
tag Label to identify the message
COMM Communicator

MPI_Recv

- Function to receive a message

MPI_Recv(*&a,n,MPI_INT,source,tag,comm,&stat*)

Where

a Data to send
n Number of elements in *a*
MPI_INT Integer data type
source ID of the sender process
tag Label to identify the message
comm Communicator
stat Array of information about the message in case of error.

Completion of the communication

Completion of the communication means that memory locations used in the message transfer can be safely accessed

- Sender side: variable sent can be reused after completion
- Receiver side: variable received can now be used

Example 2

- Compile and run mpi2.c

```
MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
                    MPI_COMM_WORLD);
//int DoSomeWork();
MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
                   MPI_COMM_WORLD, &status);
printf("I am process %d and received a message from \\
        %d\n",my_id,source);
```

- Discuss the results
- Try to send the random value calculated in Example 1

Communication Modes

Communication modes can be blocking or non-blocking

- Blocking: return from routine implies completion

- Non-Blocking: routine returns immediately, user must test for completion

Blocking Communication

MPI_Send and MPI_Recv functions are blocking

When MPI_Send returns means that the variable can be used again, for example, the content can be modified without any problem.

There is no way to know if the destination process has received the data.

Blocking Communication

MPI_Send and MPI_Recv functions are blocking

MPI_Recv returns only when all data is received in the local buffer.

Example 3

- Compile and run mpi3.c with 4 nodes
- Discuss the results
- Observe that this program uses blocking messages

Non-Blocking Communication

MPI_Isend and MPI_Irecv are non-blocking functions.

MPI_Isend(....MPI_Request *req*)
MPI_Irecv(....MPI_Request *req)*

Where:

req (output) identifier of the communications handle. It can be used later to query the status of the communication or wait for its completion.

Non-Blocking Communication

MPI_Isend and MPI_Irecv allow continue the execution of the program to do some other work.

However, it is necessary to check the completion of the communication to use the content of the buffer used to send or receive data.

The user must check manually the completion of the communication.

Non-Blocking Communication

The user has to add code to do the manual check of the communication completion.

MPI_Test(MPI_Request *req, int *flag, MPI_Status *stat)

Where:

req Communication handle
flag TRUE if the communication is completed
stat Status Object

Non-Blocking Communication

If there is not other work to do, the program must wait for the completion of the communication.

```
MPI_Wait(MPI_Request *req, MPI_Status *stat)
```

Where:

req Communication handle
stat Status Object

Synchronous Communication

MPI implementations provide some mechanisms to do syncronous communication

It means that the communication is not completed until the destination process receives the message.

MPI_Ssend	Blocking
MPI_Issend	Non-Blocking

Example 4

- Compile and run mpi4.c

```
MPI_Isend(&(buff[i]), 1, MPI_INT, 1, 111,MPI_COMM_WORLD,&(reque[i]));
MPI_Irecv(&(buff[i]),1,MPI_INT, 0, 111, MPI_COMM_WORLD,&(reque[i]));
```

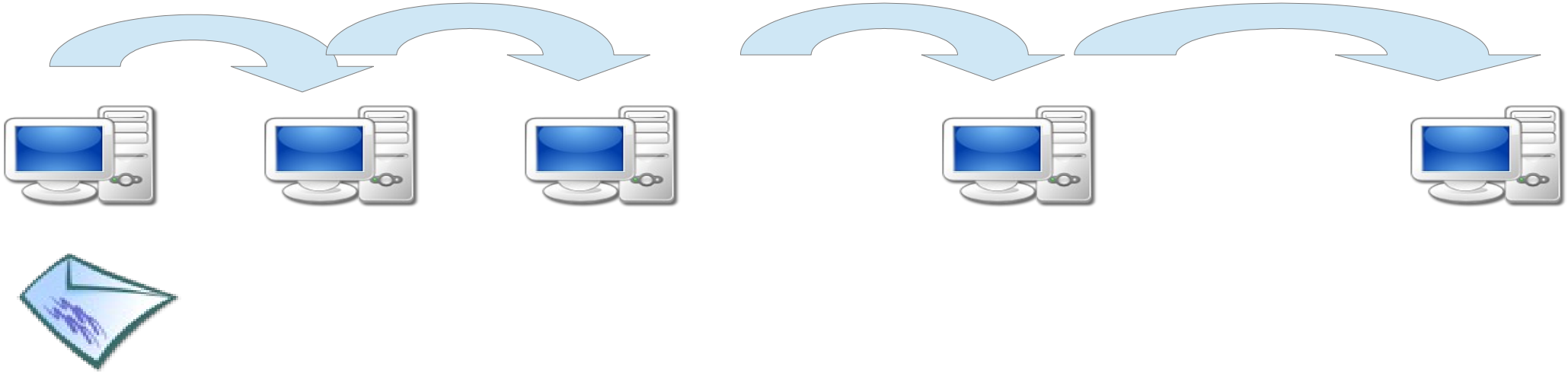
- Discuss the resutls compared to mpi3.c

Collective Communications

- MPI implementations provide functions to send/receive data to/from multiple processes.
- Collective routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations...

Collective Communications

MPI_Bcast sends the same message to all processes.



MPI_Bcast(buffer, count, datatype, src, comm)

Collective Communications

MPI_Reduce receives messages from all processes and executes a reduction operation

MPI_Reduce(*sendbuf, *recvbuf, count, type, MPI_Op op, src, comm)

Collective Communications

MPI_Op specify the type of the operation to execute on the partial results received from every process.

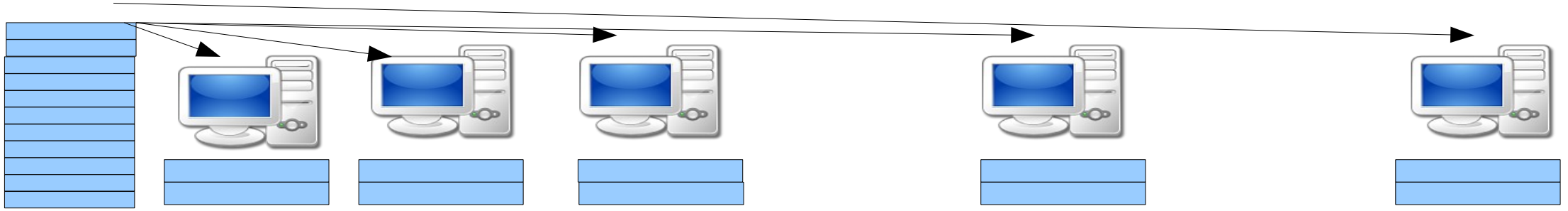
Some native functions in MPI:

- Sum
- Max
- AND, etc

The user can define new operations.

Collective Communications

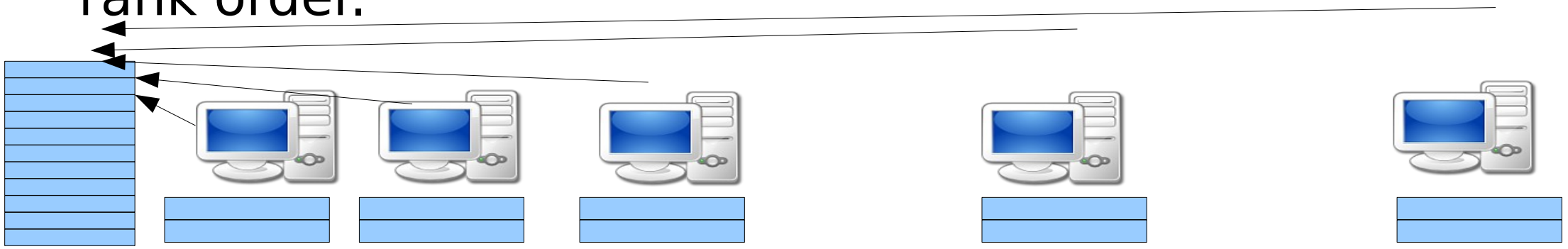
MPI_Scatter the message is split into **N** equal segments, the ith segment is sent to the ith process in the group



MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Collective Communications

MPI_Gather Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.



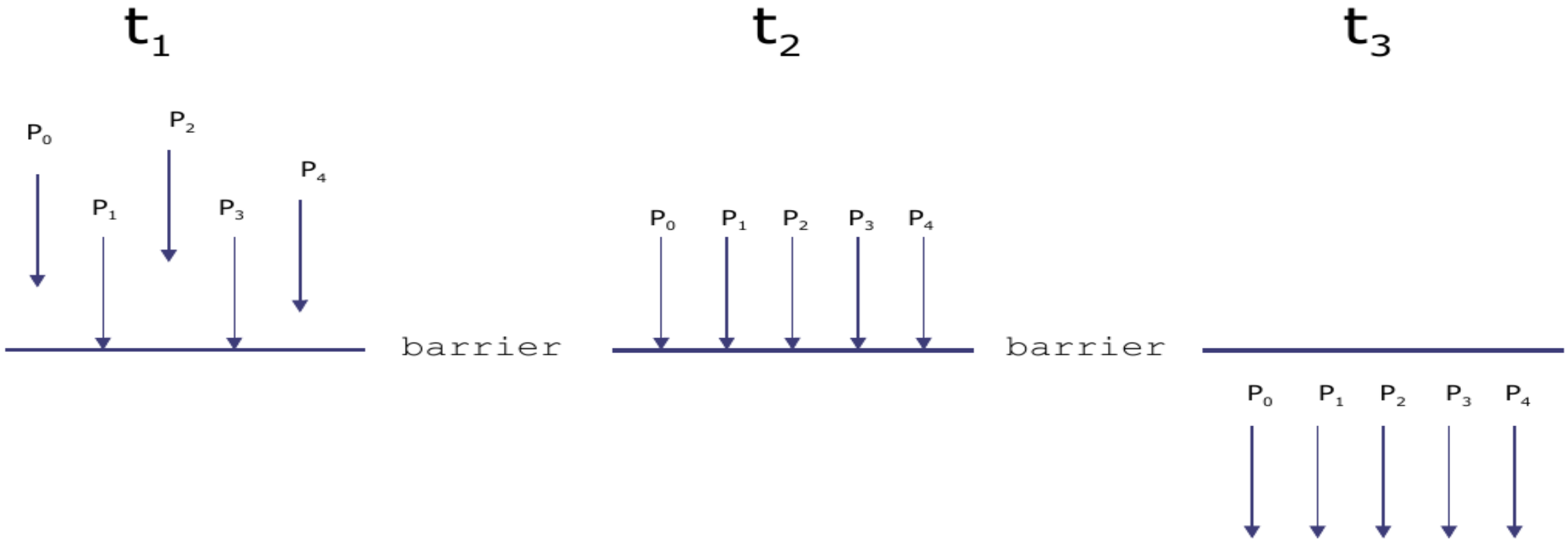
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Example 5

- Compile and run mpi5.c
- Compare with mpi3.c and mpi4.c in terms of complexity
- Use other operator, try to calculate average value of array

Barrier

MPI_Barrier(MPI_Comm comm)



CARLA 2021
LATIN AMERICA HIGH PERFORMANCE COMPUTING CONFERENCE
VIRTUAL EDITION

Thanks for taking this tutorial

Robinson Rivas-Suarez
robinson.rivas@ciens.ucv.ve

Esteban Mocskos
emocskos@dc.uba.ar

COMITTEE BOARD