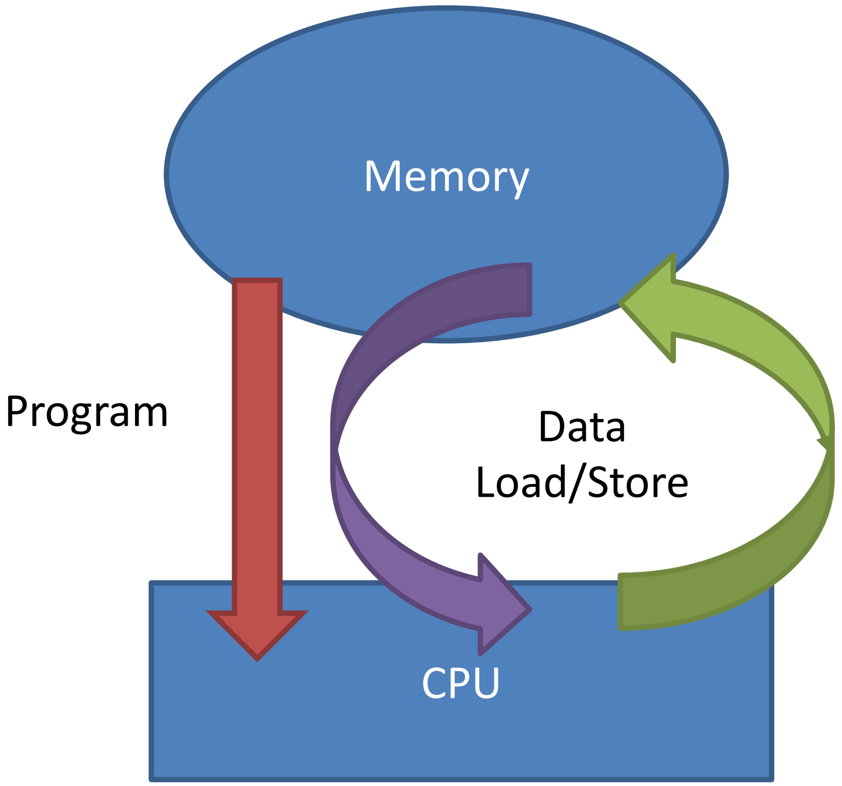
The Abdus Salam International Centre for Theoretical Physics

Parallel Programming 101

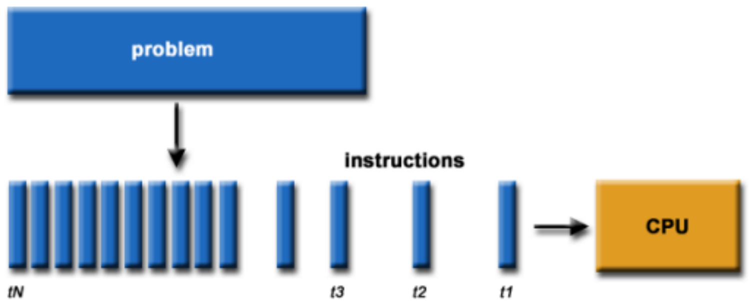
Ivan Girotto – igirotto@ictp.it

International Centre for Theoretical Physics (ICTP)

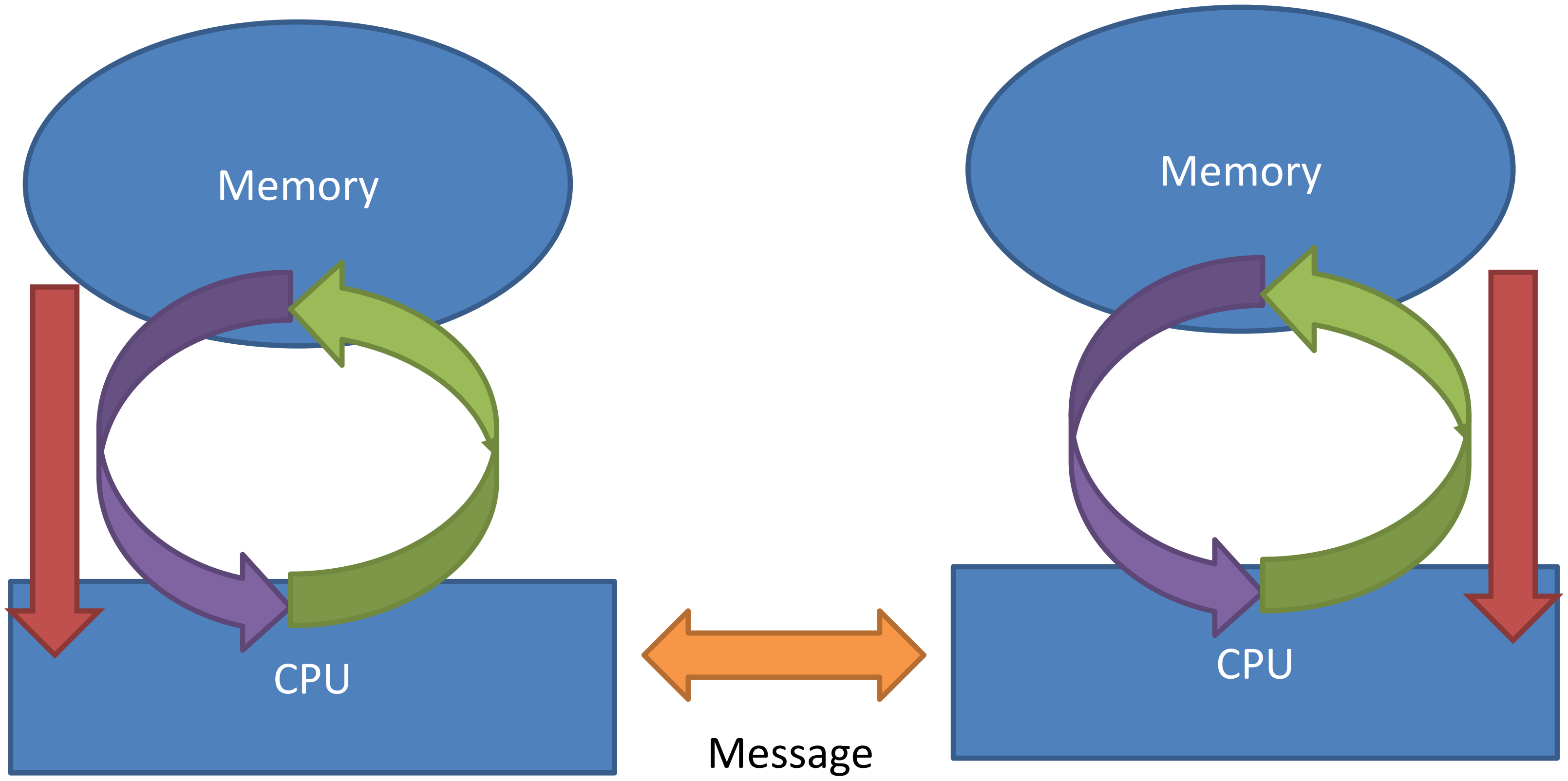
Serial Programming



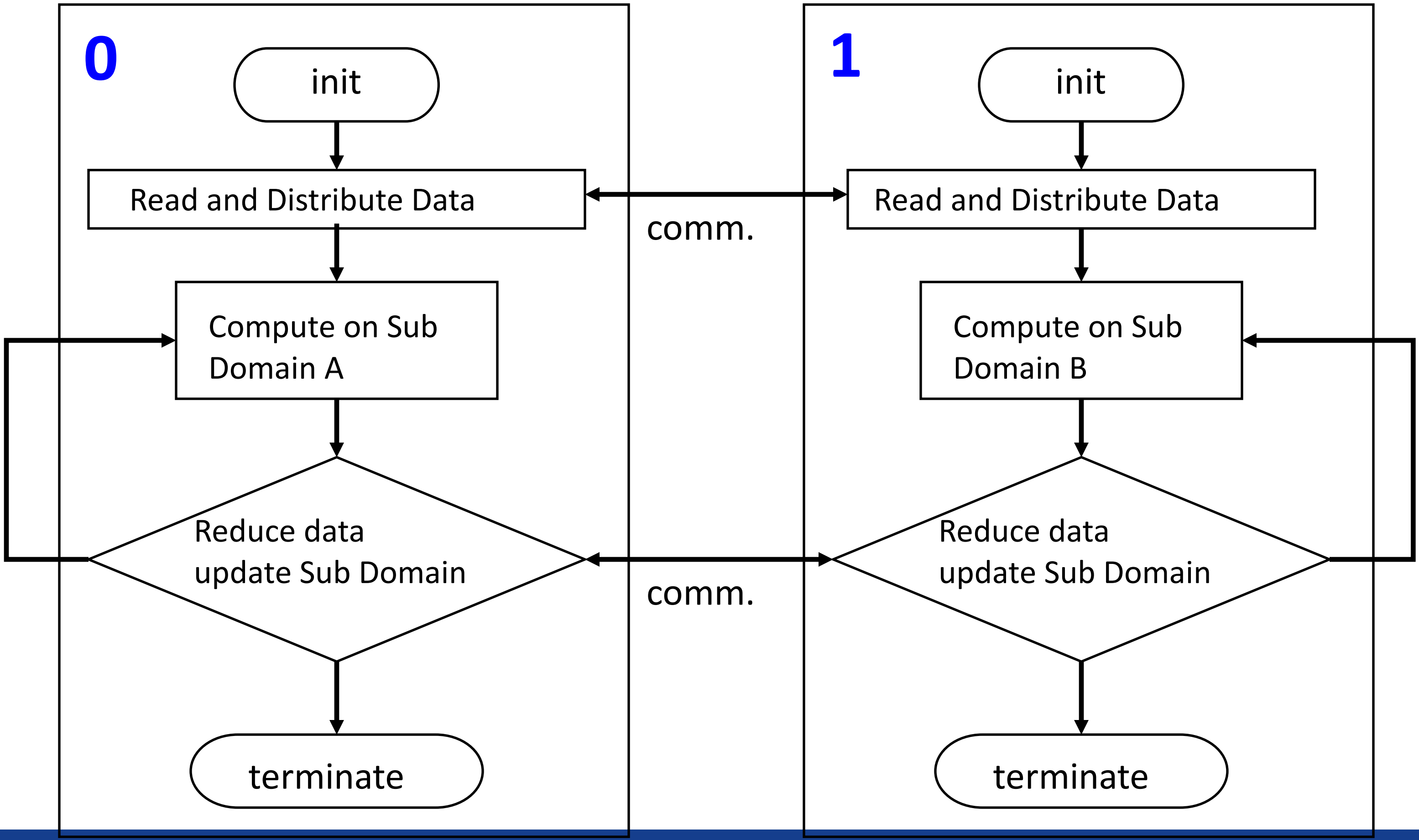
A problem is broken into a discrete series of instructions.
Instructions are executed one after another.
Only one instruction may execute at any moment in time.



Parallel Programming



What is a Parallel Program



The Abdus Salam International Centre for Theoretical Physics

MPI Program Design

- Multiple and separate processes (can be local and remote) concurrently that are coordinated and exchange data through “messages”

  => a “share nothing” parallelization
- Best for coarse grained parallelization
- Distribute large data sets; replicate small data
- Minimize communication or overlap communication and computing for efficiency

  => Amdahl's law: speedup is limited by the fraction of serial code plus communication

What is MPI?

- A standard, i.e. there is a document describing how the API are named and should behave; multiple "levels", MPI-1 (basic), MPI-2 (advanced), MPI-3 (new)
- A library or API to hide the details of low-level communication hardware and how to use it
- Implemented by multiple vendors
  - Open source and commercial versions
  - Vendor specific versions for certain hardware
  - Not binary compatible between implementations

Goals of MPI

- Allow to write software (source code) that is portable to many different parallel hardware. i.e. agnostic to actual realization in hardware
- Provide flexibility for vendors to optimize the MPI functions for their hardware
- No limitation to a specific kind of hardware and low-level communication type. Running on heterogeneous hardware is possible.
- Fortran77 and C style API as standard interface

Phases of an MPI Program

1) Startup

Parse arguments (mpirun may add some)

Identify parallel environment and rank in it

Read and distribute all data

2) Execution

Proceed to subroutine with parallel work

(can be same of different for all parallel tasks)

3) Cleanup

MPI Startup / Cleanup

Initializing the MPI environment:

CALL MPI_INIT(STATUS)

Status is integer set to MPI_SUCCESS, if operation was successful; otherwise to error code

Releasing the MPI environment:

CALL MPI_FINALIZE(STATUS)

NOTES:

All MPI tasks have to call **MPI_INIT** & **MPI_FINALIZE**

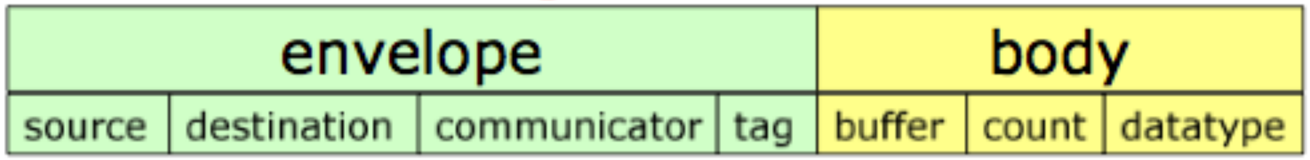
MPI_INIT may only be called once in a program

No MPI calls allowed outside of the region between calling **MPI_INIT** and **MPI_FINALIZE**

The Message

- A message is an array of elements of some particular MPI data type
- MPI defines a number of constants that correspond to language *datatypes* in Fortran and C
- When an MPI routine is called, the Fortran (or C) datatype of the data being passed must match the corresponding MPI integer constant

Message Structure



envelope				body		
source	destination	communicator	tag	buffer	count	datatype

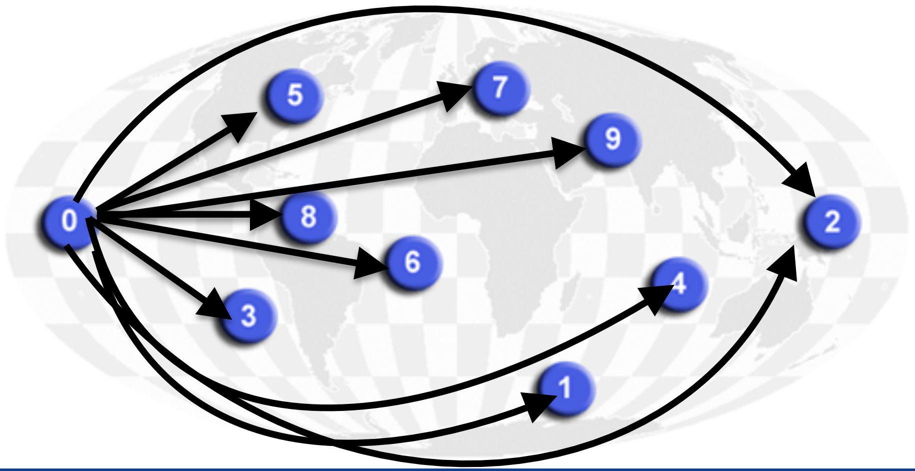
MPI in C versus MPI in Fortran

- The programming interface (“bindings”) of MPI in C and Fortran are closely related (wrappers for many other languages exist)
- MPI in C:
  - Use '#include <mpi.h>' for constants and prototypes
  - Include only once at the beginning of a file
- MPI in Fortran:
  - Use 'include “mpif.h”' for constants
  - Include at the beginning of each module
  - All MPI functions are “subroutines” with the same name and same order and type of arguments as in C with return status added as the last argument

MPI Communicators

- Is the fundamental communication facility provided by MPI library. Communication between 2 processes
- Communication take place within a communicator: Source/s and Destination/s are identified by their rank within a communicator

MPI_COMM_WORLD



Communicator Size & Process Rank

A “communicator” is a label identifying a group of processors that are ready for parallel computing with MPI

By default the **MPI_COMM_WORLD** communicator is available and contains <u>all</u> processors allocated by mpirun

<u>Size</u>: How many MPI tasks are there in total?

> **CALL MPI_COMM_SIZE(comm, size, status)**
>
> After the call the integer variable **size** holds the number of processes on the given communicator

<u>Rank</u>: What is the ID of “me” in the group?

> **CALL MPI_COMM_RANK(comm, rank, status)**
>
> After the call the integer variable **rank** holds the ID or the process. This is a number between **0** and **size-1**.

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

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

  int rank = 0; // store the MPI identifier of the process
  int npes = 1; // store the number of MPI processes

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &npes );

  fprintf( stderr, "\nI am process %d of %d MPI processes\n", rank, npes );

  MPI_Finalize();

  return 0;
}
```

Calling MPI_BCAST

MPI_BCAST(buffer, count, type, sender, comm, err)

buffer:	buffer with data
count:	number of data items to be sent
type:	type (=size) of data items
sender:	rank of sending processor of data
comm:	group identifier, MPI_COMM_WORLD
err:	error status of operation

NOTES:

buffers must be large enough (can be larger)
Data type must match (MPI does not check this)
all ranks that belong to the communicator must call this

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_FINALIZE(ierr)

end program bcast
```

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD
```

P_0

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

P_1

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

P_2

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

P_3

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
```

P_0

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

P_1

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

P_2

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

P_3

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast
  implicit none
  include "mpif.h"
  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
```

P_0

```
myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...
```

P_1

```
myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...
```

P_2

```
myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...
```

P_3

```
myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...
```

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)

  call MPI_COMM_SIZE(comm, ncpus, ierr)

  call MPI_COMM_RANK(comm, myrank, ierr)
```

P_0

myrank = 0
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

P_1

myrank = 1
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

P_2

myrank = 2
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

P_3

myrank = 3
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg
```

P_0

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

P_1

myrank = 1
ncpus = 4
imesg = 1
ierr = MPI_SUC...
comm = MPI_C...

P_2

myrank = 2
ncpus = 4
imesg = 2
ierr = MPI_SUC...
comm = MPI_C...

P_3

myrank = 3
ncpus = 4
imesg = 3
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

 implicit none

 include "mpif.h"

 integer :: myrank, ncpus, imesg, ierr
 integer, parameter :: comm = MPI_COMM_WORLD

 call MPI_INIT(ierr)
 call MPI_COMM_RANK(comm, myrank, ierr)
 call MPI_COMM_SIZE(comm, ncpus, ierr)

 imesg = myrank
 print *, "Before Bcast operation I'm ", myrank, &
     " and my message content is ", imesg

 call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)
```

P_0

```
myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_1

```
myrank = 1
ncpus = 4
imesg = 1
ierr = MPI_SUC...
comm = MPI_C...
```

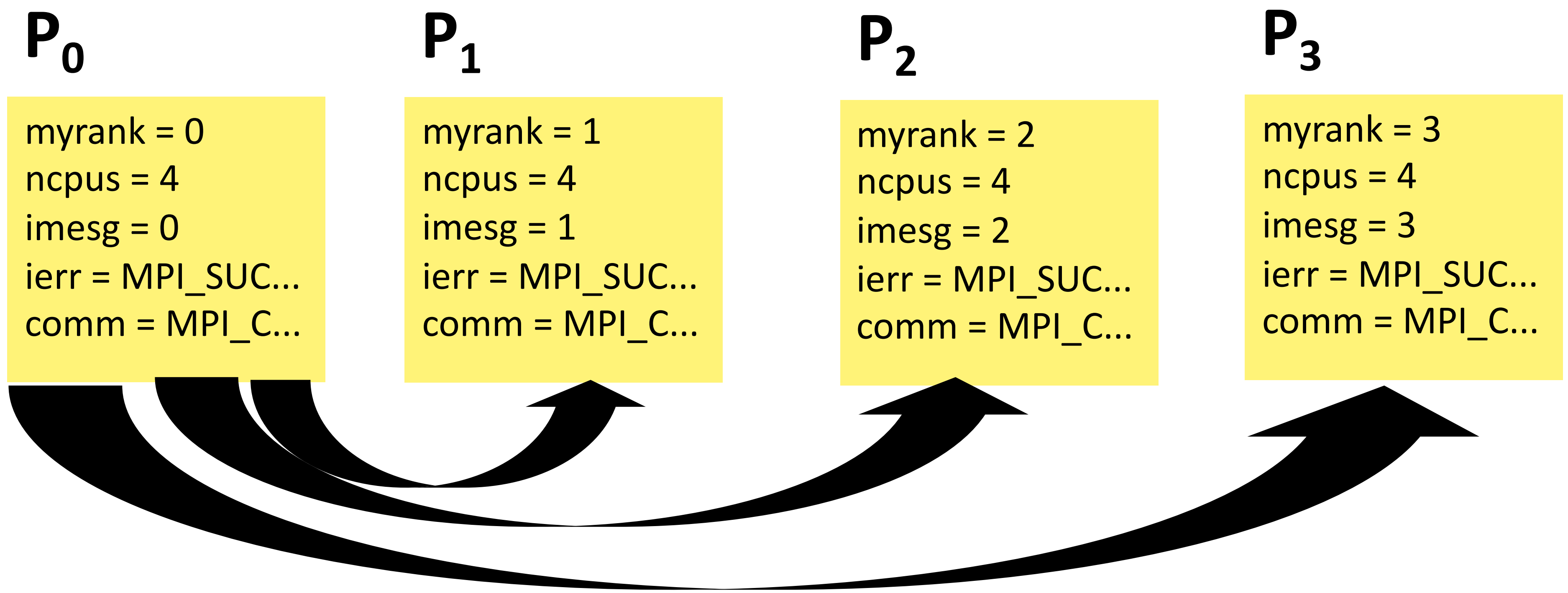
P_2

```
myrank = 2
ncpus = 4
imesg = 2
ierr = MPI_SUC...
comm = MPI_C...
```

P_3

```
myrank = 3
ncpus = 4
imesg = 3
ierr = MPI_SUC...
comm = MPI_C...
```

call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)



```
call MPI_BCAST( imesg, 1, MPI_INTEGER, 0, comm, ierr )
```

P_0

```
myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_1

```
myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_2

```
myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_3

```
myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg
```

P_0

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

P_1

myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

P_2

myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

P_3

myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_FINALIZE(ierr)
```

P_0

```
myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_1

```
myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_2

```
myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

P_3

```
myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...
```

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_FINALIZE(ierr)

end program bcast
```

P_0

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

P_1

myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

P_2

myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUCC
comm = MPI_C...

P_3

myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

The Abdus Salam International Centre for Theoretical Physics

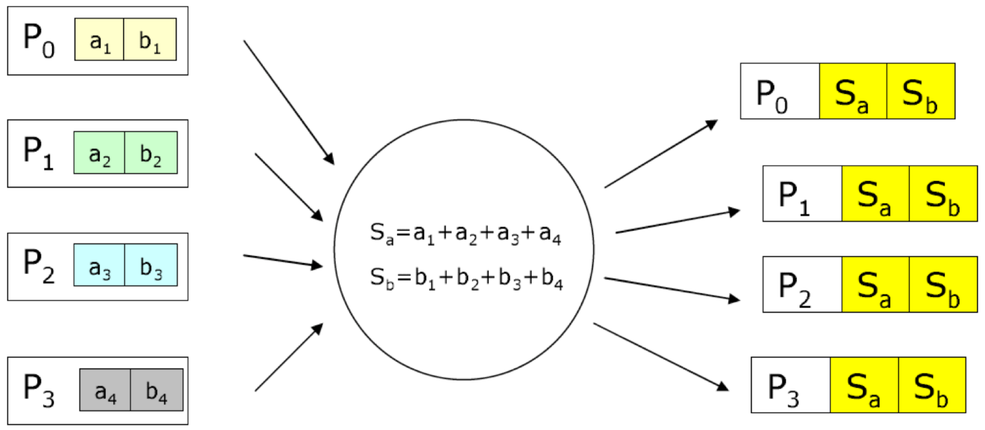
Calling MPI_REDUCE

MPI_REDUCE(in,out,count,type,op,receiver,comm,err)

in:	data to be sent (from all)
out:	storage for reduced data (on receiver)
count:	number of data items to be reduced
type:	type (=size) of data items
op:	reduction operation, e.g. **MPI_SUM**
receiver:	rank of sending processor of data
communicator:	group identifier, **MPI_COMM_WORLD**
err:	error status or **MPI_SUCCESS**

The reduction operation allow to:

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root processes
- Store the result on all processes
- **Overlap of communication and computing**



MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

The MPI_BARRIER

- Blocks until all processes have reached this routine

```
INCLUDE ’mpif.h’

MPI_BARRIER(COMM, IERROR)

 INTEGER    COMM, IERROR
```

STANDARD BLOCKING SEND - RECV

- Basic point-2-point communication routines in MPI.

MPI_SEND(buf, count, type, dest, tag, comm, ierr)

MPI_RECV(buf, count, type, dest, tag, comm, status, ierr)

Buf array of MPI type **type**.

Count (INTEGER) number of element of **buf** to be sent

Type (INTEGER) MPI type of **buf**

Dest (INTEGER) rank of the destination process

Tag (INTEGER) number identifying the message

Comm (INTEGER) communicator of the sender and receiver

Status (INTEGER) array of size **MPI_STATUS_SIZE** containing communication status information

Ierr (INTEGER) error code

Implement a program that:

- initializes on process 0 a vector of size 1GByte, all values are initialized to -*size*, where *size* is the number of processes in MPI_COMM_WORLD
- process 0 broadcast the vector to all other processes in MPI_COMM_WORLD
- All processes print the value vector[N – 1] where N is the number of element in the vector

Exercise 3:

- Compute the approximation of PI using the midpoint method (with a **REALLY** large number of rectangles)
- Reduce the final result in the last process (size – 1) and print the final output from 0! Use 101 as MPI_TAG
- Compare timing with the OpenMP version, scaling the MPI version up to 2 nodes of Ulysses (excluding operation needed for I/O)