CINECA Consorzio Interuniversitario

Parallel Programming

An overview

Carlo Cavazzoni
(High Performance Computing Group)
CINECA

CINECA Consorzio Interuniversitario

Why parallel programming?

- Solve larger problems
- Run memory demanding codes
- Solve problems with greater speed

Why on Linux clusters?

- Solve Challenging problems with low cost hardware.
- Your computer facility fit in your lab.

Modern Parallel Architectures

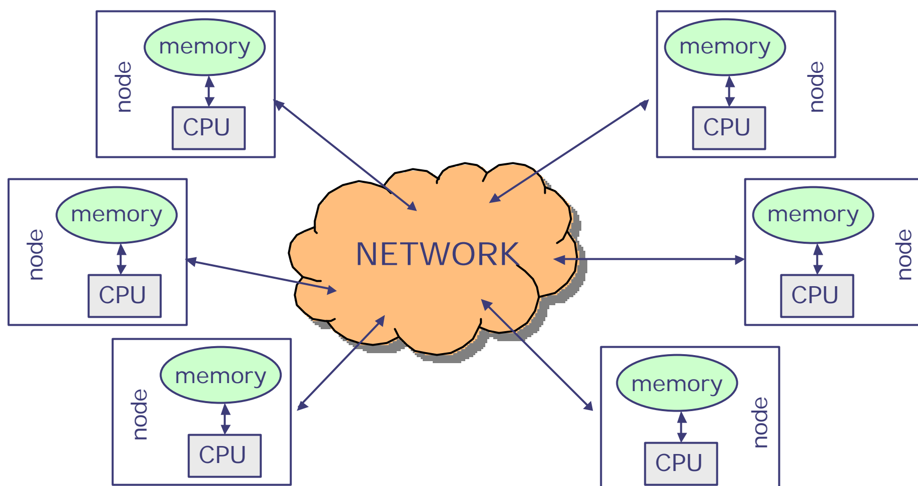
Two basic architectural scheme:

Distributed Memory

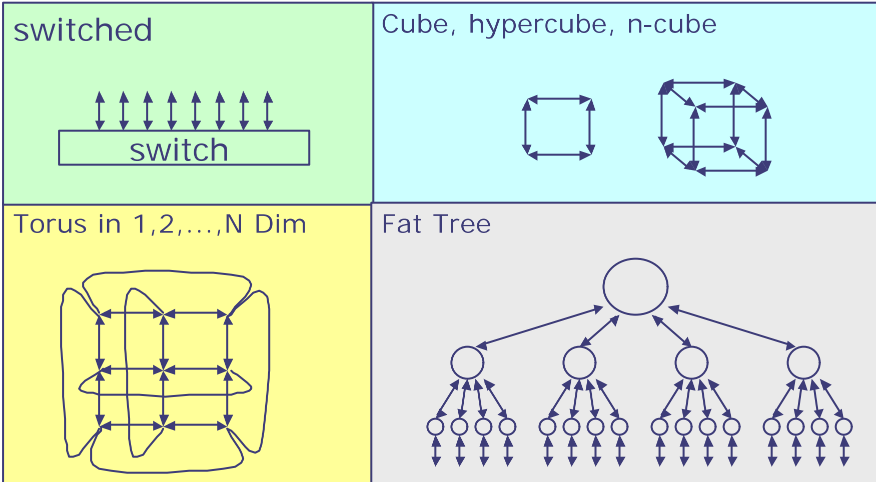
Shared Memory

Now most computers have a mixed architecture

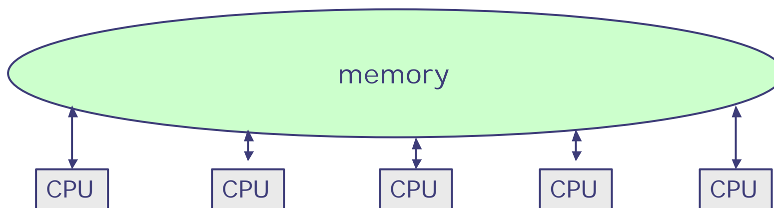
Distributed Memory



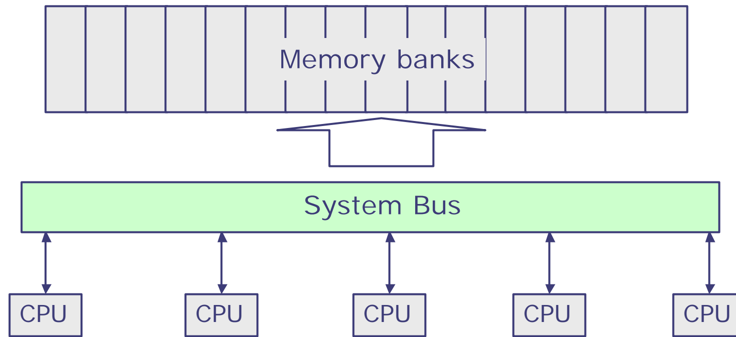
Most Common Networks



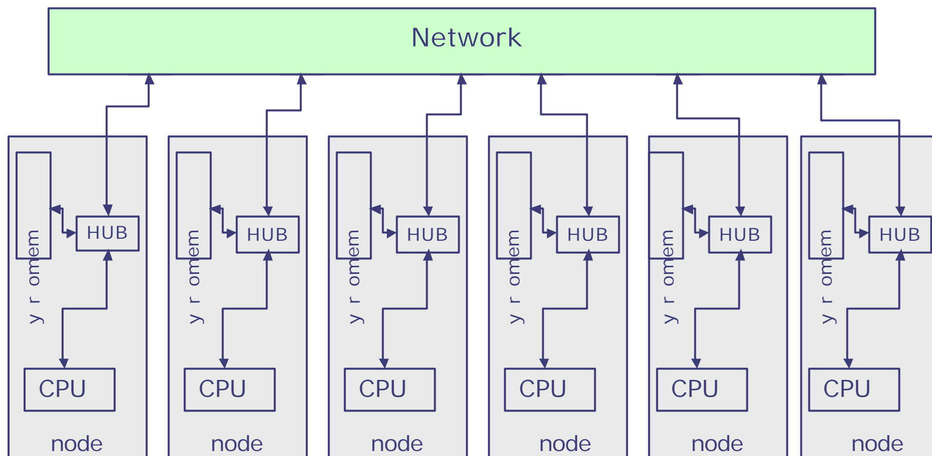
Shared Memory



Real Shared

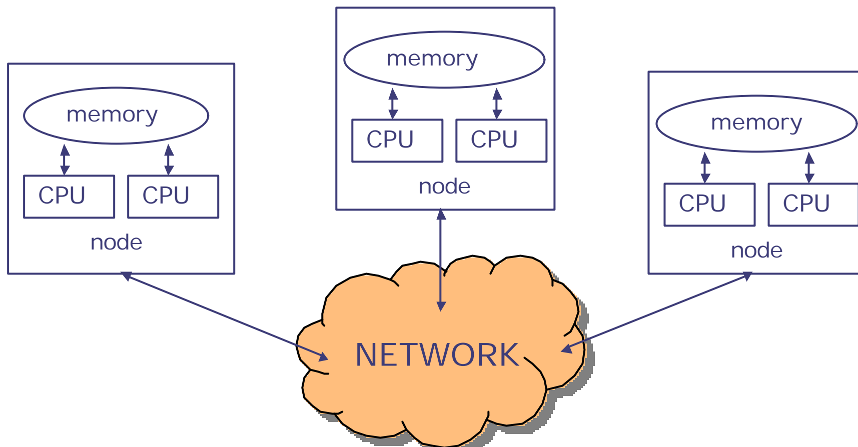


Virtual Shared



CINECA Consorzio Interuniversitario

Mixed Architectures



CINECA Consorzio Interuniversitario

Logical Machine Organization

The logical organization, seen by the programmer, could be different from the hardware architecture.

Its quite easy to logically partition a Shared Memory computer to reproduce a Distributed memory Computers.

The opposite is not true.

Parallel Programming Paradigms

The two architectures determine two basic scheme for parallel programming

Message Passing (distributed memory)
all processes could **directly** access only their local memory

Data Parallel (shared memory)
Single memory view, all processes (usually threads) could **directly** access the whole memory

Parallel Programming Paradigms, cont.

Programming Environments	
Message Passing	**Data Parallel**
Standard compilers	Ad hoc compilers
Communication Libraries	Source code Directive
Ad hoc commands to run the program	Standard Unix shell to run the program
Standards: **MPI, PVM**	Standards: **OpenMP, HPF**

Parallel Programming Paradigms, cont.

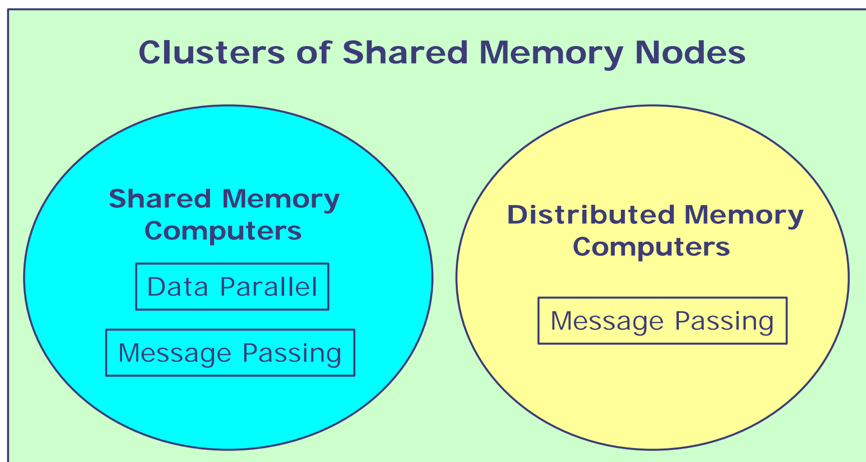
Its easy to adopt a Message Passing scheme in a Sheared Memory computers (*unix process have their private memory*).

Its less easy to follow a Data Parallel scheme in a Distributed Memory computer (*emulation of shared memory*)

Its relatively easy to design a program using the message passing scheme and implementing the code in a Data Parallel programming environments (*using OpenMP or HPF*)

Its not easy to design a program using the Data Parallel scheme and implementing the code in a Message Passing environment (*with some efforts on the T3E, shmem lib*)

Architectures vs. Paradigms



Parallel programming Models

CINECA Consorzio Interuniversitario

(again) two basic models models

Domain decomposition

Data are divided into pieces of approximately the same size and mapped to different processors. Each processors work only on its local data. The resulting code has a single flow.

Functional decomposition

The problem is decompose into a large number of smaller tasks and then the tasks are assigned to processors as they become available, Client-Server / Master-Slave paradigm.

Model	Programming Paradigms	Flint Taxonomy
Domain decomposition	Message Passing **MPI**, **PVM**	Single Program Multiple Data (**SPMD**)
	Data Parallel **HPF**	
Functional decomposition	Data Parallel **OpenMP**	Multiple Program Single Data (**MPSD**)
	Message Passing **MPI**, **PVM**	Multiple Program Multiple Data (**MPMD**)

Two basic ....

Architectures	
Distributed Memory	Shared Memory
Programming Paradigms/Environment	
Message Passing	Data Parallel
Parallel Programming Models	
Domain Decomposition	Functional Decomposition

Small important digression

When writing a parallel code, regardless of the architecture, programming model and paradigm, be always aware of

- Load Balancing
- Minimizing Communication
- Overlapping Communication and Computation

Load Balancing

Equally divide the work among the available resource: processors, memory, network bandwidth, I/O, ...

This is usually a simple task for the problem decomposition model

It is a difficult task for the functional decomposition model

CINECA Consorzio Interuniversitario

Minimizing Communication

When possible reduce the communication events:

Group lots of small communications into large one.

Eliminate synchronizations as much as possible. Each synchronization level off the performance to that of the slowest process.

CINECA Consorzio Interuniversitario

Overlap Communication and Computation

When possible code your program in such a way that processes continue to do useful work while communicating.

This is usually a non trivial task and is afforded in the very last phase of parallelization.

If you succeed, you have done. Benefits are enormous.

MPI programming model on Linux Cluster

Carlo Cavazzoni
(High Performance Computing Group)
CINECA

CINECA Consorzio Interuniversitario

INTRODUCTION: What is MPI?

MPI: Message Passing Interface

What is a message?

DATA

MPI allows data to be passed between processes

What is MPI?

MPI is standard defined in a set of documents compiled by a consortium of organizations: http://www.mpi-forum.org/

In particular the MPI documents define the APIs (application interfaces) for C, C++ and FORTRAN.

The actual implementation of the standard is demanded to the software developers of the different systems

In all systems MPI has been implemented as a library of subroutines over the network drivers and primitives

Domain decomposition and MPI

MPI is particularly suited for a Domain decomposition approach, where there is a single program flow.

Parallel computation consist of a number of processes, each working on some local data. Each process has purely local variables (no access to remote memory).

Sharing of data takes place by message passing, by explicitly sending and receiving data between processes

Goals of the MPI standard

MPI's prime goals are:
- To provide source-code portability
- To allow efficient implementation

MPI also offers:
- A great deal of functionality
- Support for heterogeneous parallel architectures

Basic Features of MPI Programs

An MPI program consists of multiple instances of a serial program that communicate by library call.

Calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and terminate communications
2. Calls used to communicate between pairs of processors. (Pair communication)
3. Calls used to communicate among groups of processors. (Collective communication)
4. Calls to create data types.

CINECA Consorzio Interuniversitario

A First Program: Hello World!

Fortran

```
PROGRAM hello

  INCLUDE 'mpif.h'
  INTEGER err

  CALL MPI_INIT(err)
  PRINT *, "hello world!"
  CALL MPI_FINALIZE(err)

END
```

C

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

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

  err = MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  err = MPI_Finalize();
}
```


CINECA Consorzio Interuniversitario

Compiling and Running MPI Programs

Compiling (**NO STANDARD**):

- specify the appropriate include directory
  (i.e. -I/mpidir/include)
- Specify the mpi library
  (i.e. -L/mpidir/lib -lmpi)
- Sometimes you may have MPI compiler wrappers that do this job for you. (i.e. mpif77)

Running (**NO STANDARD**):

- mpirun command (i.e. mpirun –np 4 myprog.x)
- Other similar command (i.e. mpiexec –n 4 myprog.x)

Basic Structures of MPI Programs

- Header files
- MPI Communicator
- MPI Function format
- Communicator Size and Process Rank
- Initializing and Exiting MPI

Header files

All Subprogram that contains calls to MPI subroutine must include the MPI header file

C:

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

Fortran:

```
include 'mpif.h'
```

The header file contains definitions of MPI constants, MPI types and functions

MPI Communicator

The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.

There is a default communicator (automatically defined):

MPI_COMM_WORLD

identify the group of all processes.

- All MPI communication subroutines have a communicator argument.
- The Programmer could define many communicator at the same time

MPI function format

C:

```
Error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

Fortran:

```
CALL MPI_XXXXX(parameter, IERROR)
```


Communicator Size and Process Rank

How many processors are associated with a communicator?
C:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran:

```
INTEGER COMM, SIZE, IERR
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

```
OUTPUT:  SIZE
```

What is the ID of a processor in a group?
C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

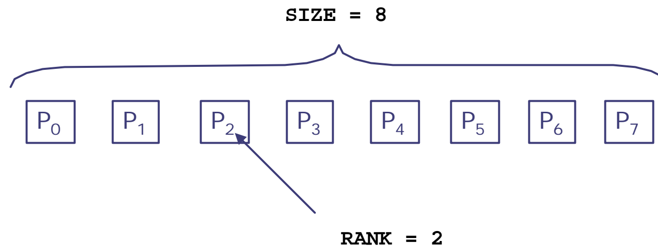
Fortran:

```
INTEGER COMM, RANK, IERR
CALL MPI_COMM_RANK(COMM, RANK, IERR)
```

```
OUTPUT:  RANK
```

CINECA Consorzio Interuniversitario

Communicator Size and Process Rank, cont.



size is the number of processors associated to the communicator

rank is the index of the process within a group associated to a communicator (**rank** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication

CINECA Consorzio Interuniversitario

Initializing and Exiting MPI

Initializing the MPI environment

C:

```
int MPI_Init(int *argc, char ***argv);
```

Fortran:

```
INTEGER IERR
CALL MPI_INIT(IERR)
```

Finalizing MPI environment

C:

```
int MPI_Finalize()
```

Fortran:

```
INTEGER IERR
CALL MPI_FINALIZE(IERR)
```

This two subprograms should be called by all process, and no other MPI calls are allowed before **mpi_init** and after **mpi_finalize**

A Template for Fortran MPI programs

```
PROGRAM template

INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      !!! INSERT YOUR PARALLEL CODE HERE !!!

CALL MPI_FINALIZE(ierr)

END
```


A Template for C MPI programs

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

void main (int argc, char * argv[])
{
  int err, nproc, myid;

  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /*** INSERT YOUR PARALLEL CODE HERE ***/

  err = MPI_Finalize();
}
```

Point to Point Communication

Let process A send a message to process B

Point to Point Communication

- Is the fundamental communication facility provided by MPI library
- Is conceptually simple: A send a message to B, B receive the message from A. It is less simple in practice.
- Communication take places within a communicator
- Source and Destination are identified by their rank in the communicator

CINECA Consorzio Interuniversitario

The Message

- A message is an array of elements of some particular MPI data type
- MPI Data types
  - Basic types
  - Derived types
- Derived type can be build up from basic types
- C types are different from Fortran types
- Messages are identified by their envelopes,
  - a message could be received only if the receiver specify the correct envelope

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

CINECA Consorzio Interuniversitario

Fortran - MPI Basic Datatypes

MPI Data type	Fortran Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	
MPI_BYTE	

C - MPI Basic Datatypes

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Definitions (Blocking and non-Blocking)

- "***Completion***" of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can now be used
- MPI communication modes differ in what conditions are needed for completion
- Communication modes can be ***blocking*** or ***non-blocking***
  - ***Blocking***: return from routine implies completion
  - ***Non-blocking***: routine returns immediately, user must test for completion

Communication Modes and MPI Subroutines

Mode	**Completion Condition**	Blocking subroutine	Non-blocking subroutine
Standard send	**Message sent (receive state unknown)**	**MPI_SEND**	**MPI_ISEND**
receive	**Completes when a message has arrived**	**MPI_RECV**	**MPI_IRECV**
Synchronous send	**Only completes when the receive has completed**	MPI_SSEND	MPI_ISSEND
Buffered send	**Always completes, irrespective of receiver**	MPI_BSEND	MPI_IBSEND
Ready send	**Always completes, irrespective of whether the receive has completed**	MPI_RSEND	MPI_IRSEND

Standard Send and Receive

basic blocking point-to-point communication routine in MPI.

Fortran:

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

Message body (buf, count, type) — Message envelope (dest, tag, comm)

Buf array of type **type** see table.
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 (if **ierr=0** no error occurs)

Standard Send and Receive

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm,
   MPI_Status *status);
```

Both in Fortran and C **MPI_RECV** accept wildcard for source (**MPI_ANYSOURCE**) and tag (**MPI_ANYTAG**)

Sending and Receiving, an example

```
PROGRAM send_recv

INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  A(1) = 3.0
  A(2) = 5.0
  CALL MPI_SEND(A, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  CALL MPI_RECV(A, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
  WRITE(6,*) myid,': a(1)=',a(1),' a(2)=',a(2)
END IF

CALL MPI_FINALIZE(ierr)
END
```

Sending and Receiving, an example

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

void main (int argc, char * argv[])
{
  int err, nproc, myid;
  MPI_Status status;
  float a[2];

  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if( myid == 0 ) {
    a[0] = 3.0, a[1] = 5.0;
    MPI_Send(a, 2, MPI_FLOAT, 1, 10, MPI_COMM_WORLD);
  } else if( myid == 1 ) {
    MPI_Recv(a, 2, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
    printf("%d: a[0]=%f a[1]=%f\n", myid, a[0], a[1]);
  }

  err = MPI_Finalize();
}
```

Again about completion

Standard MPI_RECV and MPI_SEND block the calling process until completion.
For MPI_RECV completion: the message is arrived and the process could proceed using the received data.
For MPI_SEND completion: the process could proceed and data could be overwritten without interfering with the message. But this does not mean that the message has already been sent. In many MPI implementation, depending on the message size, sending data are copied to MPI internal buffers.

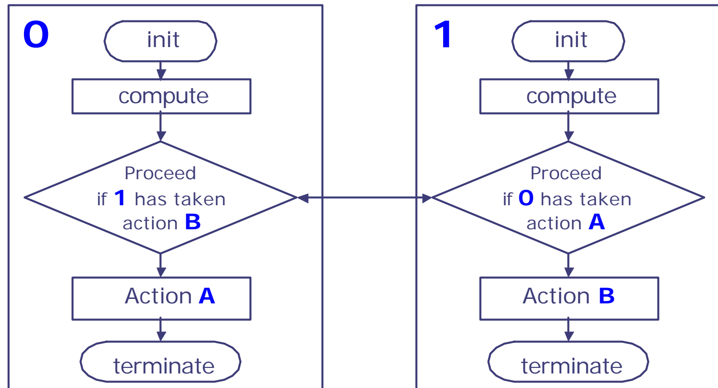
If the message is ***not buffered*** a call to MPI_SEND implies a process ***synchronization***, on the contrary this is *not true* if the message is ***buffered***.

Don't make any assumptions (implementation dependent)

CINECA Consorzio Interuniversitario

DEADLOCK

Deadlock occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.



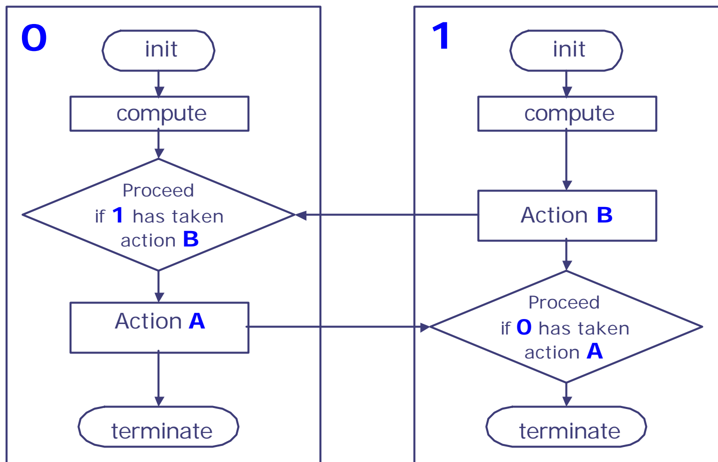
CINECA Consorzio Interuniversitario

Simple DEADLOCK

```
PROGRAM deadlock
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
END IF
WRITE(6,*) myid, ': a(1)=', a(1), ' a(2)=', a(2)
CALL MPI_FINALIZE(ierr)
END
```

CINECA Consorzio Interuniversitario

Avoiding DEADLOCK



CINECA Consorzio Interuniversitario

Avoiding DEADLOCK

```
PROGRAM avoid_lock
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ': a(1)=', a(1), ' a(2)=', a(2)
CALL MPI_FINALIZE(ierr)
END
```

DEADLOCK: the most common error

```
PROGRAM error_lock
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ': a(1)=', a(1), ' a(2)=', a(2)
CALL MPI_FINALIZE(ierr)
END
```


Non-Blocking Send and Receive

Non-Blocking communications allows the separation between the initiation of the communication and the completion.

Advantages: between the initiation and completion the program could do other useful computation (latency hiding).

Disadvantages: the programmer has to insert code to check for completion.

Non-Blocking Send and Receive

Fortran:

```
MPI_ISEND(buf, count, type, dest, tag, comm, req, ierr)
MPI_IRECV(buf, count, type, dest, tag, comm, req, ierr)
```

buf array of type **type** see table.
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
req (INTEGER) output, identifier of the communications handle
ierr (INTEGER) output, error code (if **ierr=0** no error occurs)

Non-Blocking Send and Receive

C:

```
int MPI_Isend(void *buf, int count,
  MPI_Datatype type, int dest, int tag,
  MPI_Comm comm, MPI_Request *req);

int MPI_Irecv (void *buf, int count,
  MPI_Datatype type, int dest, int tag,
  MPI_Comm comm, MPI_Request *req);
```

Waiting and Testing for Completion

Fortran:
MPI_WAIT(req, status, ierr)

A call to this subroutine cause the code to wait until the communication pointed by req is complete.

req (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).
Status (INTEGER) array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.
ierr (INTEGER) output, error code (if **ierr=0** no error occours).

C:
```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
```

Waiting and Testing for Completion

Fortran:
MPI_TEST(req, flag, status, ierr)

A call to this subroutine sets **flag** to **.true.** if the communication pointed by **req** is complete, sets **flag** to **.false.** otherwise.

req (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).
Flag (LOGICAL) output, **.true.** if communication **req** has completed **.false.** otherwise
Status (INTEGER) array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.
ierr (INTEGER) output, error code (if **ierr=0** no error occours).

C:
```
int MPI_Wait(MPI_Request *req, int *flag, MPI_Status *status);
```

Send and Receive, the easy way.

The easiest way to send and receive data without warring about deadlocks

Fortran:

Sender side: sndbuf, snd_size, snd_type, destid, ip

```
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type, destid, ip,
rcvbuf, rcv_size, rcv_type, sourid, ip, comm, status, ierr)
```

Receiver side: rcvbuf, rcv_size, rcv_type, sourid, ip

Send and Receive, the easy way.

```
PROGRAM send_recv
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_SENDRECV(a, 2, MPI_REAL, 1, 10, b, 2, MPI_REAL, 1, 11,
   MPI_COMM_WORLD, status, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SENDRECV(a, 2, MPI_REAL, 0, 11, b, 2, MPI_REAL, 0, 10,
   MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
```

Lab 1: My First MPI program

Implement and test the code:

1. Implements the Template MPI program
2. Compile
3. Run
4. Insert some code in the template (printout rank and size)

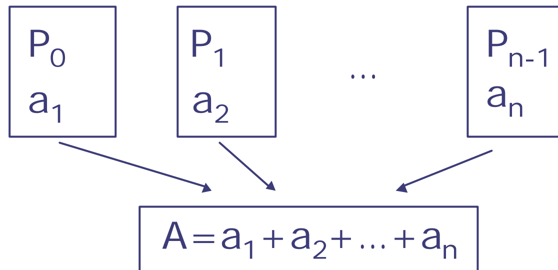
Lab 1: DEADLOCKS

Implement and test the code:

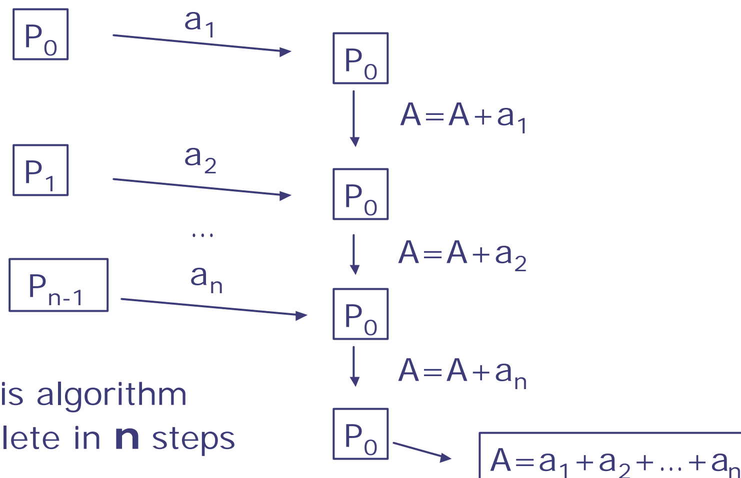
1. The Deadlock program
2. The Avoid Deadlock program
3. The Deadlock program with non-blocking **MPI_ISEND**, **MPI_IRECV**, **MPI_WAIT** and **MPI_TEST**
4. The Most common error program with **MPI_SEND**, **MPI_RECV** and arrays of increasing size

LAB 2: Reduction and Binary Tree

Reduction: sum up the partial results of different process (maybe the most common parallel operation required in a parallel program)

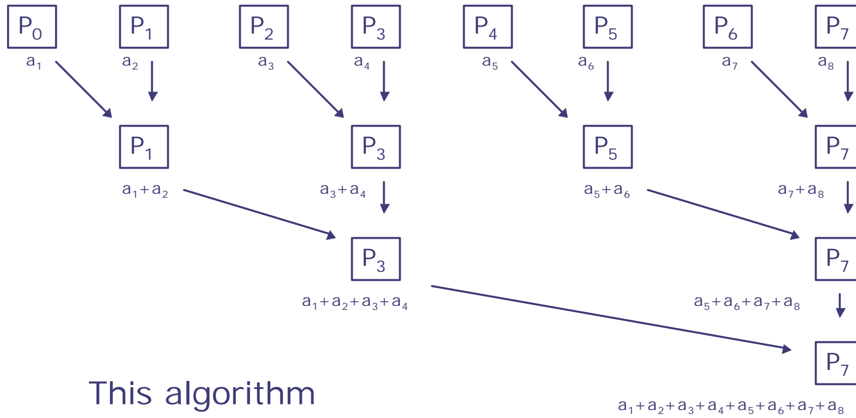


LAB 2: A Simple strategy



This algorithm complete in **n** steps

LAB 2: Binary Tree - I



This algorithm complete in $\mathbf{log_2 n}$ steps

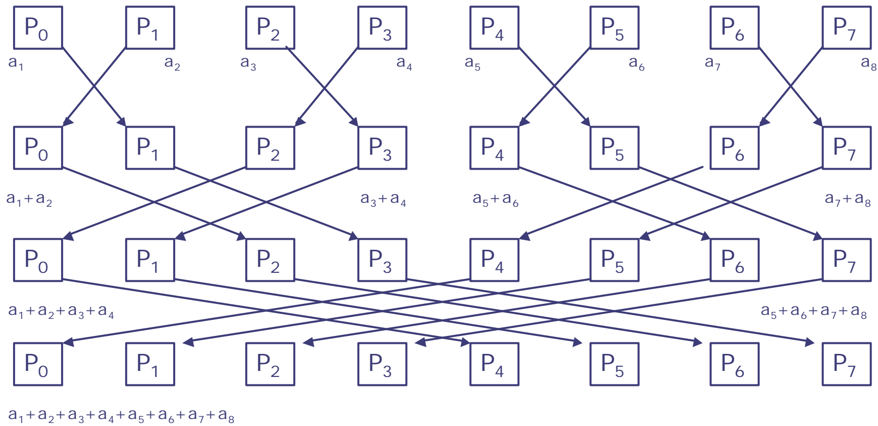
Lab 2: Binary Tree - I

As an hint observe that:

	Sender	Receiver
Step 1	`MOD(myid,2)=0`	`MOD(myid,2)=1`
Step 2	`MOD(myid,4)=1`	`MOD(myid,4)=3`
Step 3	`MOD(myid,8)=3`	`MOD(myid,8)=7`
…		
Step n	`MOD(myid,2**n)= 2**(n-1)-1`	`MOD(myid,2**n)= 2**n-1`

`myid`: processor index

Lab 2: Binary Tree - II



Lab 2: Binary Tree - II

As an hint observe that:

	Sender	Receiver
Step 1	`MOD(myid,2)/1=0`	`MOD(myid,2)/1=1`
Step 2	`MOD(myid,4)/2=0`	`MOD(myid,4)/2=1`
Step 3	`MOD(myid,8)/4=0`	`MOD(myid,8)/4=1`
…		
Step n	`MOD(myid,2**n)/ 2**(n-1)=0`	`MOD(myid,2**n)/ 2**(n-1)=1`

`myid: processor index`

Lab 2: Parallel Sum

Implement the parallel sum:

1. Using Simple strategy
2. Binary tree I
3. Binary tree II

Use only **MPI_SEND** and **MPI_RECV**

Collective Communications

The power of MPI

Collective Communications

-Communications involving a group of process
-Called by ***all*** processes in a communicator

- Barrier Synchronization
- Broadcast
- Gather/Scatter
- Reduction (sum, max, prod, ...)

CINECA Consorzio Interuniversitario

Characteristics

- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size

Safest communication mode

MPI_Barrier

Stop processes until all processes within a communicator reach the barrier

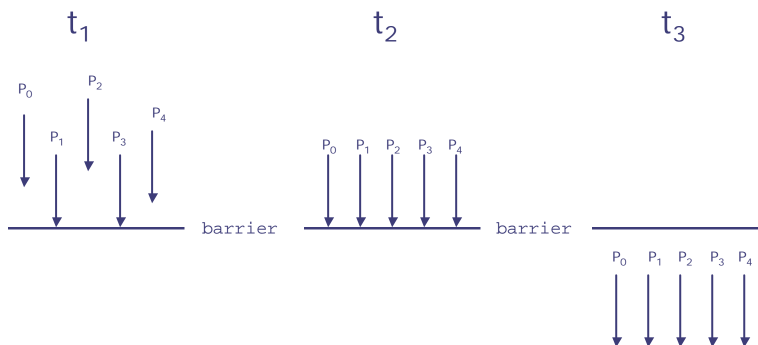
Fortran:

```
CALL MPI_BARRIER( comm, ierr)
```

C:

```
int MPI_Barrier(MPI_Comm comm)
```

Barrier



CINECA Consorzio Interuniversitario

Broadcast (MPI_BCAST)

One-to-all communication: same data sent from root process to all others in the communicator

Fortran:

```
INTEGER count, type, root, comm, ierr
CALL MPI_BCAST(buf, count, type, root, comm, ierr)
```

Buf array of type **type**

C:

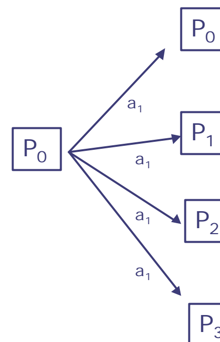
```
int MPI_Bcast(void *buf, int count, MPI_Datatype
  datatypem int root, MPI_Comm comm)
```

All processes must specify same **root**, **rank** and **comm**

CINECA Consorzio Interuniversitario

Broadcast

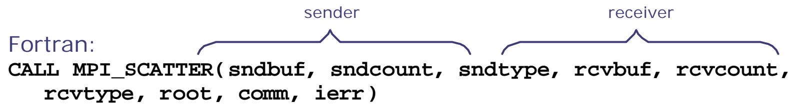
```
PROGRAM broad_cast
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 IF( myid .EQ. 0 ) THEN
   a(1) = 2.0
   a(2) = 4.0
 END IF
 CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_FINALIZE(ierr)
 END
```



CINECA Consorzio Interuniversitario

MPI_Scatter

One-to-all communication: different data sent from root process to all others in the communicator



Fortran:

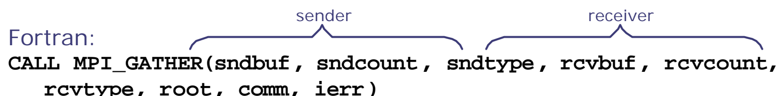
```
CALL MPI_SCATTER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount,
   rcvtype, root, comm, ierr)
```

- Arguments definition are like other MPI subroutine
- **sndcount** is the number of elements sent to each process, not the size of **sndbuf**, that should be **sndcount** times the number of process in the communicator
- The sender arguments are significant only at root

CINECA Consorzio Interuniversitario

MPI_Gather

One-to-all communication: different data collected by the root process, from all others processes in the communicator. Is the opposite of Scatter



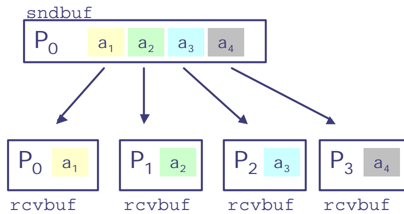
Fortran:

```
CALL MPI_GATHER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount,
   rcvtype, root, comm, ierr)
```

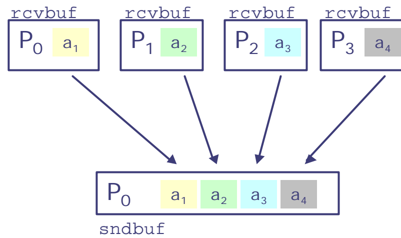
- Arguments definition are like other MPI subroutine
- **rcvcount** is the number of elements collected from each process, not the size of **rcvbuf**, that should be **rcvcount** times the number of process in the communicator
- The receiver arguments are significant only at root

Scatter/Gather

Scatter



Gather



Scatter/Gather examples

scatter

```
PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, nsnd, I, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(16), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
IF( myid .eq. root ) THEN
  DO i = 1, 16
    a(i) = REAL(i)
  END DO
END IF
nsnd = 2
CALL MPI_SCATTER(a, nsnd, MPI_REAL, b, nsnd,
& MPI_REAL, root, MPI_COMM_WORLD, ierr)
WRITE(6,*) myid, ': b(1)=', b(1), 'b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
```

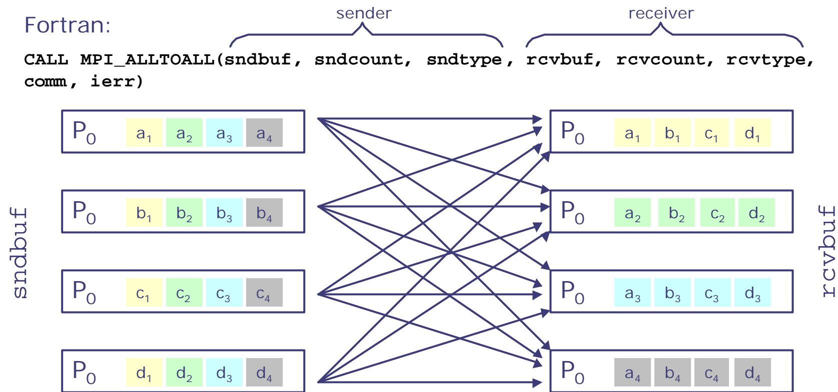
gather

```
PROGRAM gather
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, nsnd, I, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(16), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
b(1) = REAL( myid )
b(2) = REAL( myid )
nsnd = 2
CALL MPI_GATHER(b, nsnd, MPI_REAL, a, nsnd,
&  MPI_REAL, root MPI_COMM_WORLD, ierr)
IF( myid .eq. root ) THEN
  DO i = 1, (nsnd*nproc)
    WRITE(6,*) myid, ': a(i)=', a(i)
  END DO
END IF
CALL MPI_FINALIZE(ierr)
END
```

MPI_Alltoall

Fortran:

```
CALL MPI_ALLTOALL(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype,
comm, ierr)
```



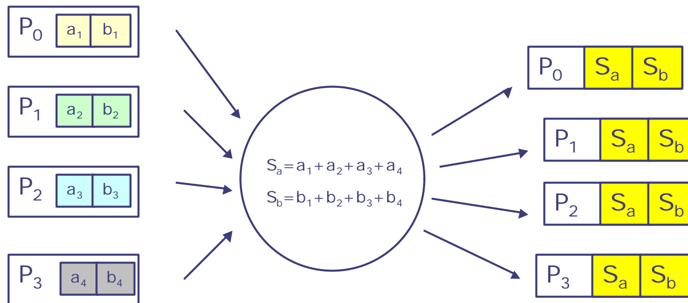
Very useful to implement data transposition

Reduction

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

Reduce, Parallel Sum



Reduction function works with arrays

other operation: product, min, max, and, ...

Internally is usually implemented with a binary tree

MPI_REDUCE and MPI_ALLREDUCE

Fortran:

MPI_REDUCE(snd_buf, rcv_buf, count, type, op, root, comm, ierr)

`snd_buf`	input array of type `type` containing local values.
`rcv_buf`	output array of type `type` containing global results
`count`	`(INTEGER)` number of element of `snd_buf` and `rcv_buf`
`type`	`(INTEGER)` MPI type of `snd_buf` and `rcv_buf`
`op`	`(INTEGER)` parallel operation to be performed
`root`	`(INTEGER)` MPI id of the process storing the result
`comm`	`(INTEGER)` communicator of processes involved in the operation
`ierr`	`(INTEGER)` output, error code (if `ierr=0` no error occours)

MPI_ALLREDUCE(snd_buf, rcv_buf, count, type, op, comm, ierr)

The argument `root` is missing, the result is stored to all processes.

Predefined Reduction Operations

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

Reduce, cont.

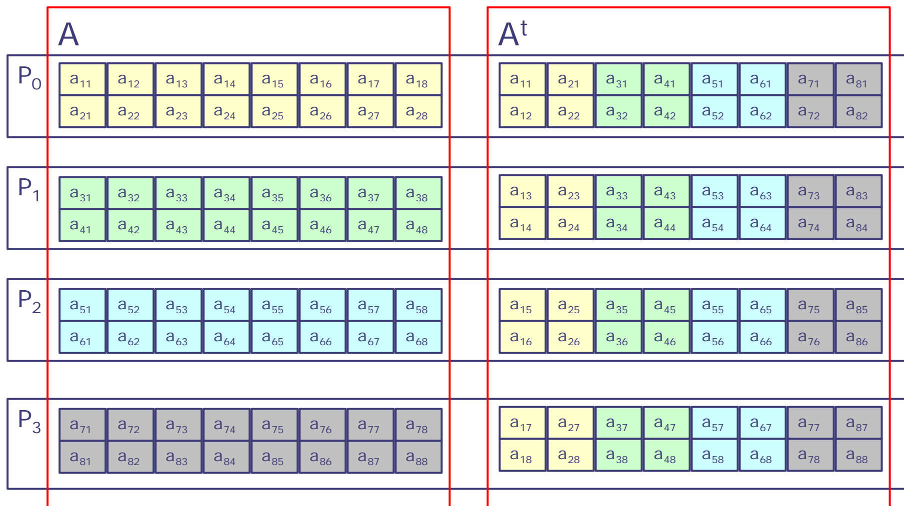
C:

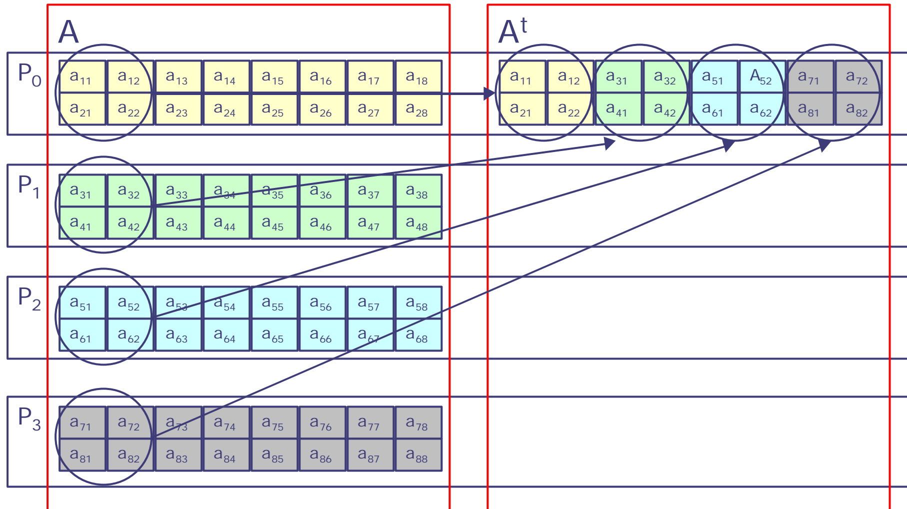
```
int MPI_Reduce(void * snd_buf, void * rcv_buf, int count,
   MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(void * snd_buf, void * rcv_buf, int count,
   MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

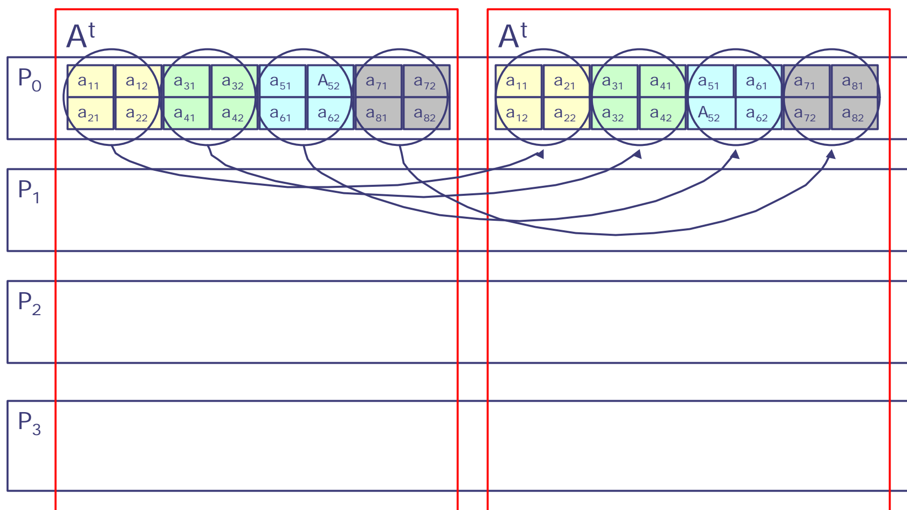
Reduce, example

```
PROGRAM reduce
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), res(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
a(1) = 2.0
a(2) = 4.0
CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
& MPI_COMM_WORLD, ierr)
IF( myid .EQ. 0 ) THEN
  WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
END IF
CALL MPI_FINALIZE(ierr)
END
```


Lab 3, Matrix transposition, elements distribution




CINECA Consorzio Interuniversitario
Lab3: Step 1, communicate blocks
A
At
P0
P1
P2
P3
a11 a12 a13 a14 a15 a16 a17 a18
a21 a22 a23 a24 a25 a26 a27 a28
a31 a32 a33 a34 a35 a36 a37 a38
a41 a42 a43 a44 a45 a46 a47 a48
a51 a52 a53 a54 a55 a56 a57 a58
a61 a62 a63 a64 a65 a66 a67 a68
a71 a72 a73 a74 a75 a76 a77 a78
a81 a82 a83 a84 a85 a86 a87 a88
a11 a12 a31 a32 a51 A52 a71 a72
a21 a22 a41 a42 a61 a62 a81 a82
January 31 - February 15 2002
MPI programming model, ICTP - Linux Cluster School
MPI programming model - 93


CINECA Consorzio Interuniversitario
Lab 3: Step 2, transpose each blocks
At
At
P0
P1
P2
P3
a11 a12 a31 a32 a51 A52 a71 a72
a21 a22 a41 a42 a61 a62 a81 a82
a11 a21 a31 a41 a51 a61 a71 a81
a12 a22 a32 a42 A52 a62 a72 a82
January 31 - February 15 2002
MPI programming model, ICTP - Linux Cluster School
MPI programming model - 94

Lab 3: Matrix transposition

Implement the Transposition algorithm
Using:

1. Multiple gather or scatter operation
2. A single alltoall communication

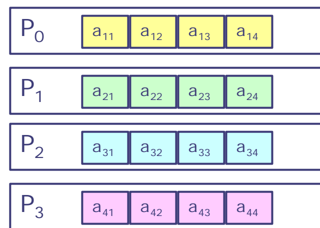
Lab 4: Parallel Matrix Multiplication

Write a subprogram implementing matrix multiplication

$$C = A\,B \longrightarrow c_{ij} = \Sigma_k\, a_{ik} b_{kj}$$

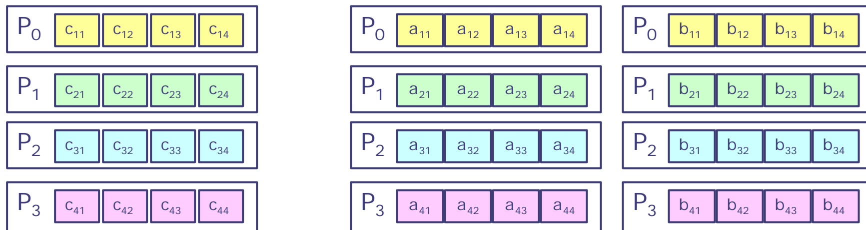
A, B and C being NxN matrixes distributed by row across processes

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}



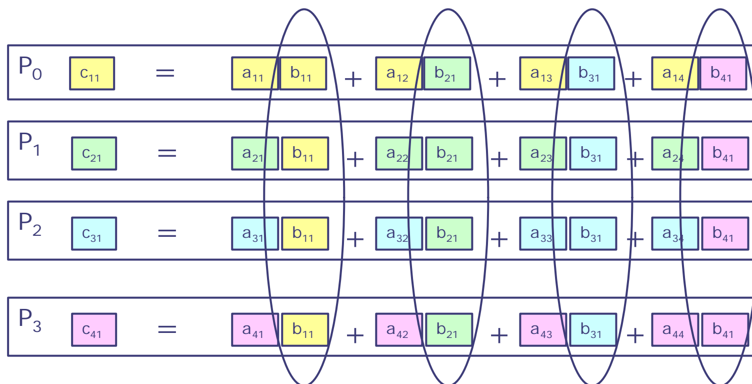
Lab 4: Parallel Matrix Multiplication

C = A B



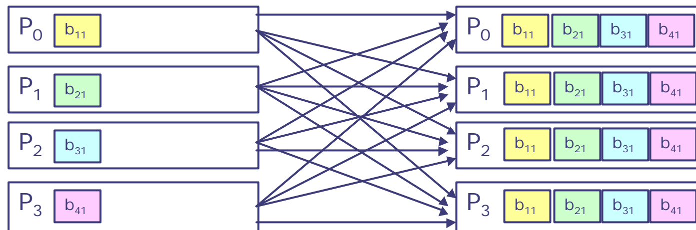
$c_{11} = a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31} + a_{14} b_{41}$

Lab 4: Parallel Matrix Multiplication



Lab 4: Step 1, alltoall

Perform an All gather, of the first column of slements or blocks



Lab 4: Step 2, local work

Each processor calculate the first elements or blocks of the matrix C

P_0: $c_{11} = a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31} + a_{14} b_{41}$

P_1: $c_{21} = a_{21} b_{11} + a_{22} b_{21} + a_{23} b_{31} + a_{24} b_{41}$

P_2: $c_{31} = a_{31} b_{11} + a_{32} b_{21} + a_{33} b_{31} + a_{34} b_{41}$

P_3: $c_{41} = a_{41} b_{11} + a_{42} b_{21} + a_{43} b_{31} + a_{44} b_{41}$

Lab 4: Step 3, local work

Repeat Step 1 and Step 2 for each column elements or blocks of matrix C, until matrix C is complete

Lab 2: solution

```
PROGRAM simple_reduce
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, i
INTEGER status(MPI_STATUS_SIZE)
REAL a, ra, sump

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
DO i = 1, nproc
  IF( myid .EQ. i-1 ) a = REAL( i )
END DO
IF( myid .EQ. 0 ) sump = a
DO i = 2, nproc
  IF( myid .EQ. 0 ) THEN
    CALL MPI_RECV(ra, 1, MPI_REAL, i-1, i, MPI_COMM_WORLD, status, ierr)
    sump = sump + ra
  ELSE IF( myid .EQ. i-1 ) THEN
    CALL MPI_SEND(a, 1, MPI_REAL, 0, i, MPI_COMM_WORLD, ierr)
  END IF
END DO

IF( myid .EQ. 0 ) WRITE(6,*) myid, ': sum = ', sump
CALL MPI_FINALIZE(ierr)
END
```

Lab 2: solution

```
      PROGRAM binary_reduce1
      INCLUDE 'mpif.h'
      INTEGER ierr, myid, nproc, i, inm, in2, in2m, n
      INTEGER status(MPI_STATUS_SIZE)
      REAL a, ra, sump

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      DO i = 1, nproc
        IF( myid .EQ. i-1 ) a = REAL( i )
      END DO

      sump = a
      i = 1
10    IF( i .GE. nproc ) GO TO 20
        in2  = i * 2
        IF( MOD(myid,in2) .EQ. (i-1) ) THEN
          CALL MPI_SEND(sump, 1, MPI_REAL, myid+i, i, MPI_COMM_WORLD,
     &      ierr)
        ELSE IF( MOD(myid,in2) .EQ. (in2-1) ) THEN
          CALL MPI_RECV(ra, 1, MPI_REAL, myid-i, i, MPI_COMM_WORLD,
     &      status, ierr)
          sump = sump + ra
        END IF
        i    = i * 2
        GO TO 10
20    CONTINUE

      IF( myid .EQ. nproc-1 ) WRITE(6,*) myid, ': sum = ', sump
      CALL MPI_FINALIZE(ierr)
      END
```


Lab 2: solution

```
      PROGRAM binary_reduce2
      INCLUDE 'mpif.h'
      INTEGER ierr, myid, nproc, i, in2
      INTEGER status(MPI_STATUS_SIZE)
      REAL a, ra, sump
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      DO i = 1, nproc
        IF( myid .EQ. i-1 ) a = REAL( i )
      END DO
      sump = a
      i = 1
10    IF( i .GE. nproc ) GO TO 20
        in2  = i * 2
        IF( MOD(myid,in2)/i .EQ. 0 ) THEN
          CALL MPI_SEND(sump, 1, MPI_REAL, myid+i, i, MPI_COMM_WORLD,
     &      ierr)
          CALL MPI_RECV(ra, 1, MPI_REAL, myid+i, i, MPI_COMM_WORLD,
     &      status, ierr)
          sump = sump + ra
        ELSE IF( MOD(myid,in2)/i .EQ. 1 ) THEN
          CALL MPI_RECV(ra, 1, MPI_REAL, myid-i, i, MPI_COMM_WORLD,
     &      status, ierr)
          CALL MPI_SEND(sump, 1, MPI_REAL, myid-i, i, MPI_COMM_WORLD,
     &      ierr)
          sump = sump + ra
        END IF
        i    = i * 2
        GO TO 10
20    CONTINUE
      WRITE(6,*) myid, ': sum = ', sump
      CALL MPI_FINALIZE(ierr)
      END
```

MPI advanced Topics

MPI Virtual Topologies

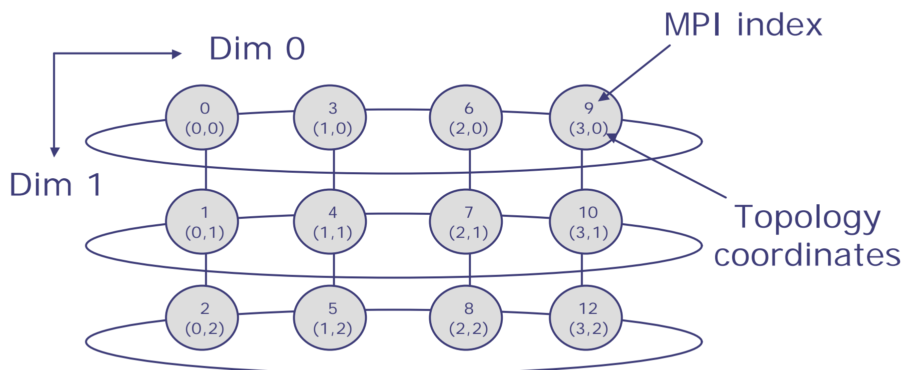
- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code
- Can allow MPI to optimize communications

Use Virtual Topologies

- Create new communicators
- Compute the processes coordinates
- Mapping functions

CINECA Consorzio Interuniversitario

Virtual Topology an Example 2D Torus



Topology types

- **Cartesian topologies**
  - Each process is connected to its neighbors in a virtual grid
  - Boundaries can be cyclic
  - Processes can be identified by Cartesian coordinates
- **Graph topologies**
  - General graphs
  - Will not be covered here

Creating a Cartesian Virtual Topology

C:

```
int MPI_Cart_create (MPI_Comm comm_old, int ndims, int
   *dims, int *periods, int reorder, MPI_Comm
   *comm_cart)
```

Fortran:

```
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
CALL MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,
   REORDER, COMM_CART, IERROR)
```

Arguments

`comm_old`	(input) existing communicator
`Ndims`	(input) number of dimensions
`periods`	(input) logical array indicating whether a dimension is cyclic (If **TRUE**, cyclic boundary conditions)
`reorder`	(input) logical (If **FALSE**, rank preserved) (If **TRUE**, possible rank reordering)
`comm_cart`	(output) new cartesian communicator

Mapping process grid coordinates to ranks

C:

```
int MPI_Cart_rank (MPI_Comm comm, init *coords,
   int *rank)
```

Fortran:

```
INTEGER COMM,COORDS(*),RANK,IERROR
CALL MPI_CART_RANK(COMM,COORDS,RANK,IERROR)
```

Mapping ranks to process grid coordinates

C:

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int
   maxdims, int *coords)
```

Fortran:

```
INTEGER COMM,RANK,MAXDIMS,COORDS(*),IERROR
CALL MPI_CART_COORDS(COMM,RANK,MAXDIMS,COORDS,IERROR)
```

Virtual Topology example

```
#include<mpi.h>
/* Run with 12 processes */
void main(int argc, char *argv[]) {
  int rank;
  MPI_Comm vu;
  int dim[2],period[2],reorder;
  int coord[2],id;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  dim[0]=4; dim[1]=3;
  period[0]=TRUE; period[1]=FALSE;
  reorder=TRUE;
  MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
  if(rank==5){
    MPI_Cart_coords(vu,rank,2,coord);
    printf("P:%d My coordinates are %d %d\n",rank,coord[0],coord[1]);
  }
  if(rank==0) {
    coord[0]=3; coord[1]=1;
    MPI_Cart_rank(vu,coord,&id);
    printf("The processor at position (%d, %d) has rank %d\n",coord[0],coord[1],id);
  }
  MPI_Finalize();
}
```

Virtual Topology example

```
      PROGRAM Cartesian
C     Run with 12 processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer vu,dim(2),coord(2),id
      logical period(2),reorder
        CALL MPI_INIT(err)
        CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
        CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
        dim(1)=4
        dim(2)=3
        period(1)=.true.
        period(2)=.false.
        reorder=.true.
        call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
        if(rank.eq.5) then
          call MPI_CART_COORDS(vu,rank,2,coord,err)
          print*,'P:',rank,' my coordinates are',coord
        end if
        if(rank.eq.0) then
          coord(1)=3
          coord(2)=1
          call MPI_CART_RANK(vu,coord,id,err)
          print*,'P:',rank,' processor at position',coord,' is',id
        end if
        CALL MPI_FINALIZE(err)
      END
```


Computing ranks of neighboring processes

C:

```
int MPI_Cart_shift (MPI_Comm comm, int direction, int
   disp, int *rank_source, int *rank_dest)
```

Fortran:

```
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST
INTEGER IERR
CALL MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
RANK_DEST, IERR)
```

CINECA Consorzio Interuniversitario

Arguments

`direction`	dimension in which the shift should be made
`disp`	length of the shift in processor coordinates (+ or -)
`rank_source`	where calling process should receive a message from during the shift
`rank_dest`	where calling process should send a message to during the shift

Does not actually shift data: returns the correct ranks for a shift which can be used in subsequent communication calls

If shift off of the topology, MPI_Proc_null is returned

CINECA Consorzio Interuniversitario

Cartesian Partitioning

Often we want to do an operation on only part of an existing Cartesian topology

Cut a grid up into 'slices'

A new communicator is produced for each slice

Each slice can then perform its own collective communications

MPI_Cart_sub and MPI_CART_SUB generate new communicators for the slice

MPI_Cart_sub

C:

```
int MPI_Cart_sub (MPI_Comm comm, int *remain_dims,
MPI_Comm *newcomm)
```

Fortran:

```
INTEGER COMM,NEWCOMM,IERROR
LOGICAL REMAIN_DIMS(*)
CALL MPI_CART_SUB(COMM,REMAIN_DIMS,NEWCOMM,IERROR)
```

If **`comm`** is a 2x3x4 grid and **`remain_dims={TRUE,FALSE,TRUE}`**, then three new communicators are created each being a 2x4 grid

Calling processor receives back only the new communicator it is in

MPI on the web

http://oscinfo.osc.edu/training/

http://www.netlib.org/mpi/index.html

http://www-unix.mcs.anl.gov/mpi/learning.html

http://www.ncsa.uiuc.edu/UserInfo/Training/