MPI programming model

Gian Franco Marras
g.marras@cineca.it
Dipartimento Supercalcolo, Applicazioni e Innovazione (SCAI)
CINECA - High Performance System Group
Casalecchio di Reno (BO) – Italy
www.cineca.it - www.cineca.it/en/index.htm

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

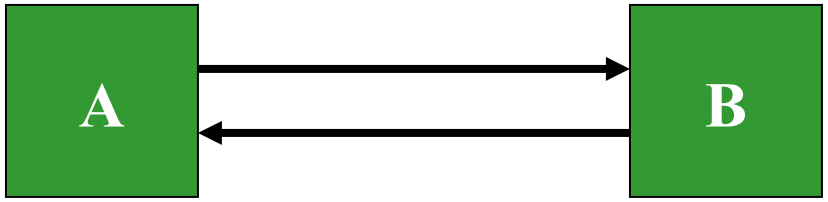
In all systems MPI has been implemented as a library of subroutines over the network drivers and primitives

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. (Point-to-point communication);
3. Calls used to communicate among groups of processors. (Collective communication);
4. Calls to create data types.

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=Returns the size of the group associated with a communicator.

What is the ID of a processor in a group?

C:

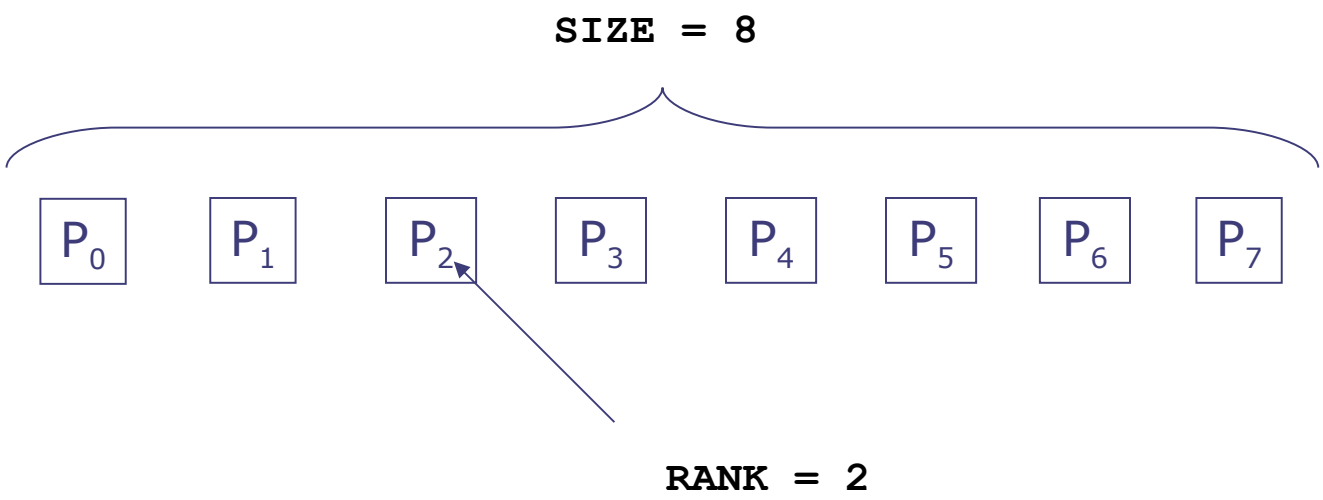
```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

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

OUTPUT: RANK = Determines the rank of the calling process in the communicator.

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

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 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();
}
```

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

The Message

- A message is an array of elements of some particular MPI data type
- 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

  - buffer: initial address of send buffer
  - Count: number of elements send
  - Source: rank of source
  - Destination: rank of destination
  - Tag : message tag
  - Datatype: datatype of each send/recv buffer element

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

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	

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, sour, tag, comm, status, ierr)
```

Message body — Message envelope

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
Sour (INTEGER) rank of the source 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 sour, 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.

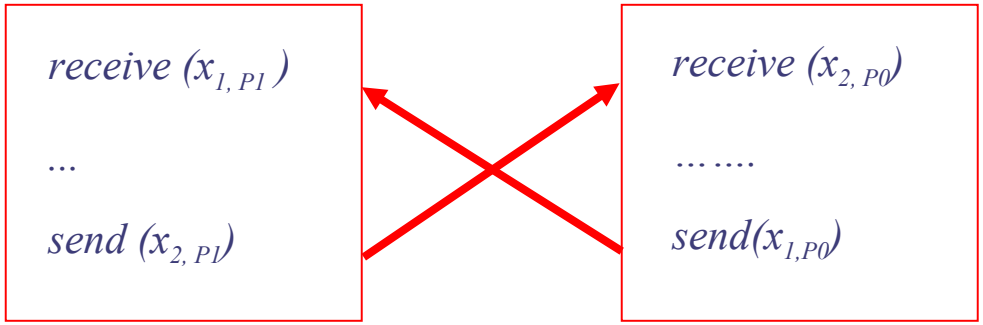
For MPI_SEND completion: the process could proceed and data could be overwritten without interfering with the message. But this does not mean that the message has already been sent. In many MPI implementation, depending on the message size, sending data are copied to MPI internal buffers.

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)

DEADLOCK

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



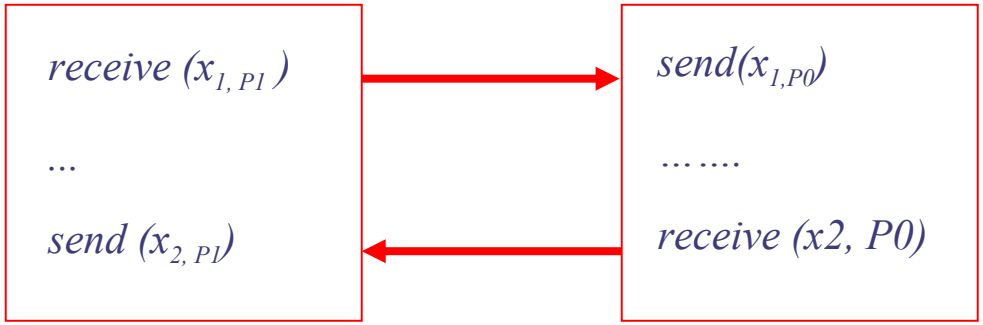
Simple DEADLOCK

```
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
```

Avoiding DEADLOCK



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, sour, 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
sour	(INTEGER)	rank of the source 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 sour, 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);
```

Example: ISEND & WAIT

```
PROGRAM async
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, status(MPI_STATUS_SIZE),req
  INTEGER, PARAMETER :: ncount=20000
  REAL A(ncount), B(ncount)
  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_ISEND(a, ncount, MPI_REAL, 1, 10, MPI_COMM_WORLD, req, ierr)
  ELSE IF( myid .EQ. 1 ) THEN
    CALL MPI_IRECV(b, ncount, MPI_REAL, 0, 10, MPI_COMM_WORLD, req, ierr)
  END IF
  CALL MPI_WAIT(req, status, ierr)
  IF( myid .EQ. 1 ) THEN
    WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
  ENDIF
  CALL MPI_FINALIZE(ierr)
END
```

Send and Receive, the easy way.

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

Fortran:

Sender side

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

Receiver side

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
```

Collective Communications

Collective Communications

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

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

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

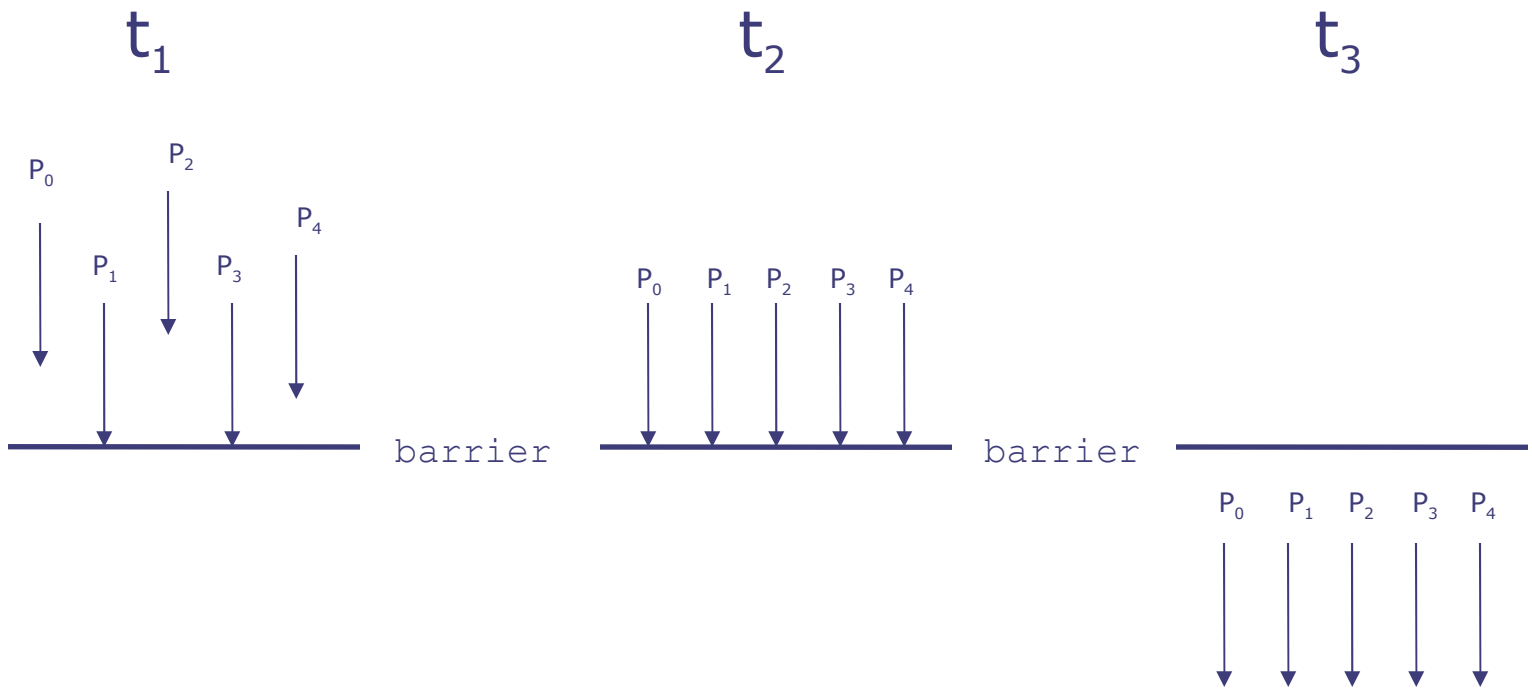
Fortran:

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

C:

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

Barrier



MPI_BARRIER blocks the caller until all group members have called it.

Broadcast (MPI_BCAST)

One-to-all communication: MPI_BCAST send a message from the process with rank root to all processes of the group, itself included.

Fortran:

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

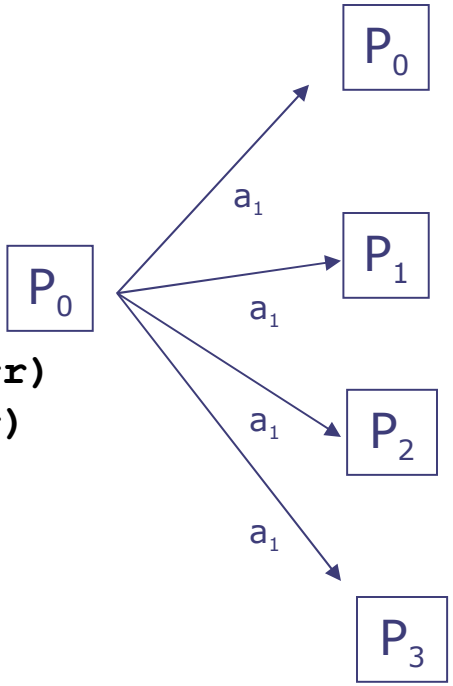
C:

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

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

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
```



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.

sender receiver

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

MPI_Scatter

the root sends a message. This message is split into n equal segments, the i-th segment is sent to the i-th process in the group, and each process receives this message.

Fortran:

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

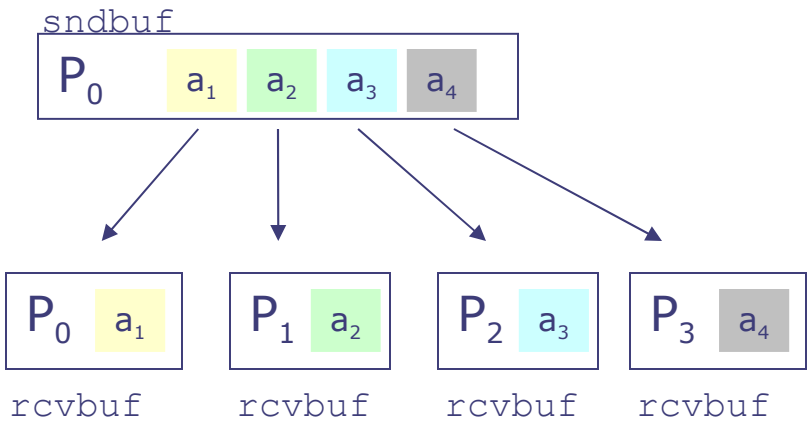
sender: sndbuf, sndcount, sndtype

receiver: rcvbuf, rcvcount, rcvtype

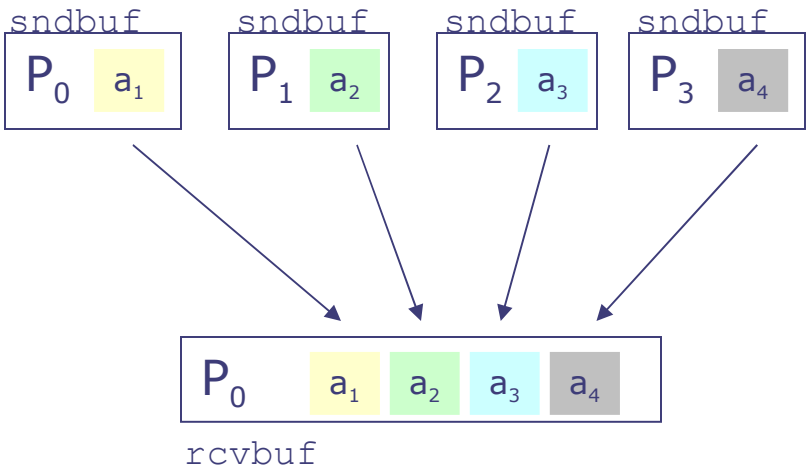
- Arguments definition are like other MPI subroutine
- **sndcount** is the number of elements sent to each process, not the size of **sndbuf**, that should be **sndcount** times the number of process in the communicator
- The sender arguments are significant only at root

Scatter/Gather

Scatter



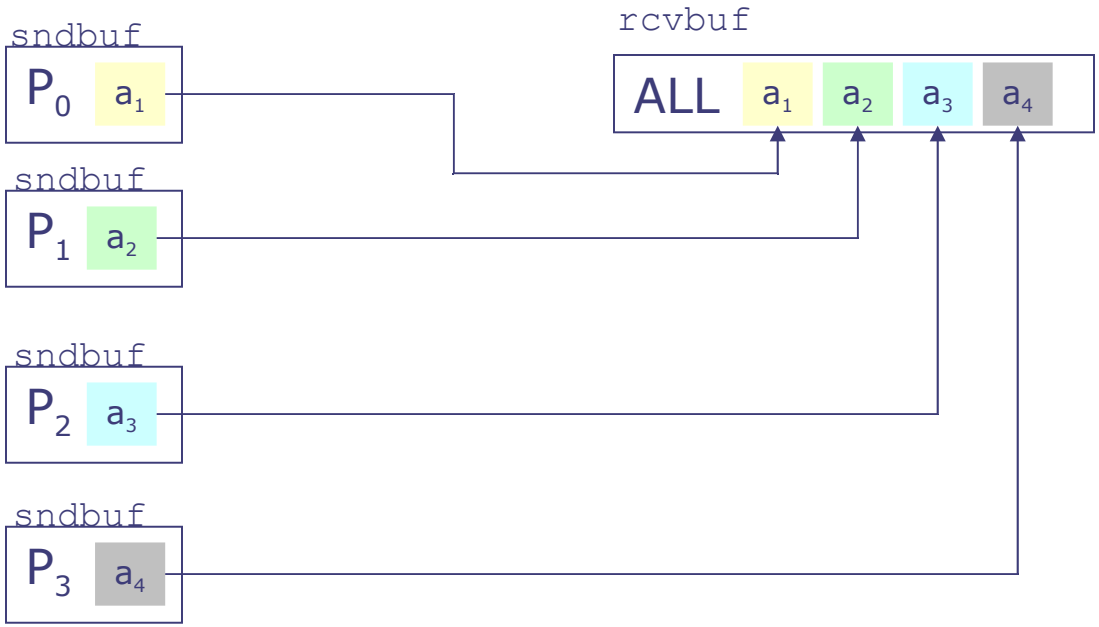
Gather



All Gather

MPI_ALLGATHER can be thought of as MPI_GATHER,
but where all processes receive the result, instead of just the root.
(MPI_GATHER + MPI_BCAST)

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



Scatter/Gather examples

scatter

```
 PROGRAM scatter
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, nsnd, i
 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

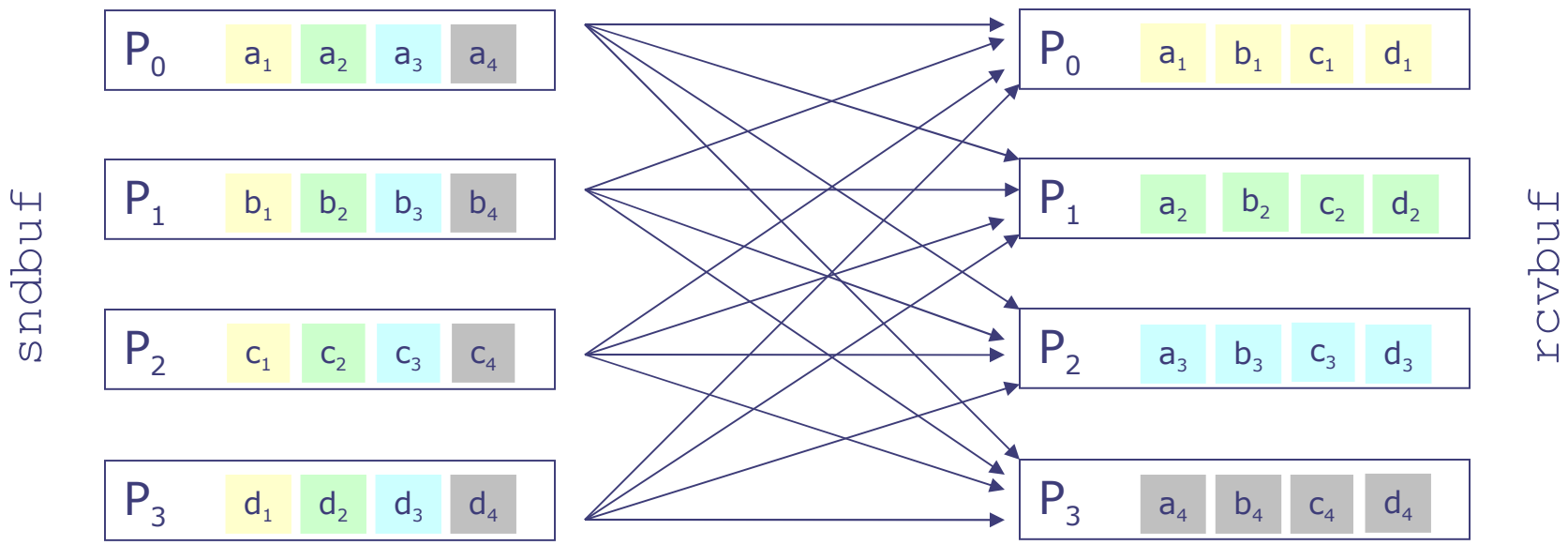
```
 PROGRAM gather
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, nsnd, i
 REAL A(16), B(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 b(1) = REAL( myid )
 b(2) = REAL( myid )
 nsnd = 2
 CALL MPI_GATHER(b, nsnd, MPI_REAL, 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:

sender receiver

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



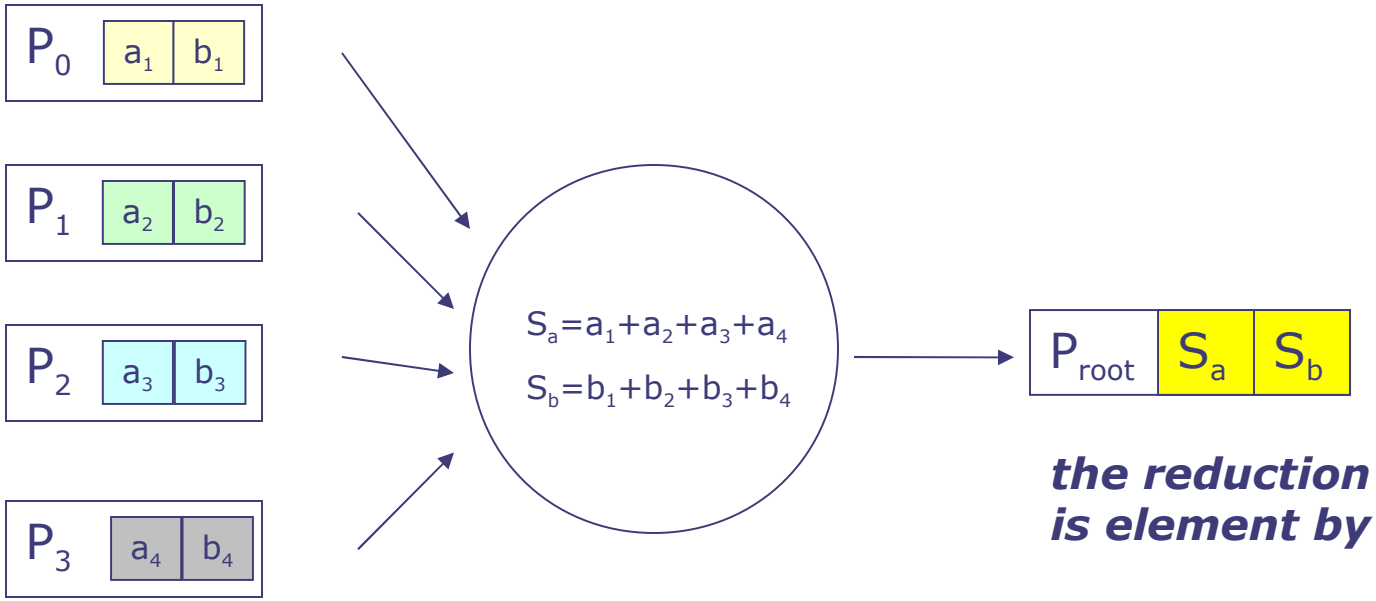
Very useful to implement data transposition

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



the reduction of an array is element by element!

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, example

```
PROGRAM reduce
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, root
  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
```

MPI on the web

http://www.mpi-forum.org/

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

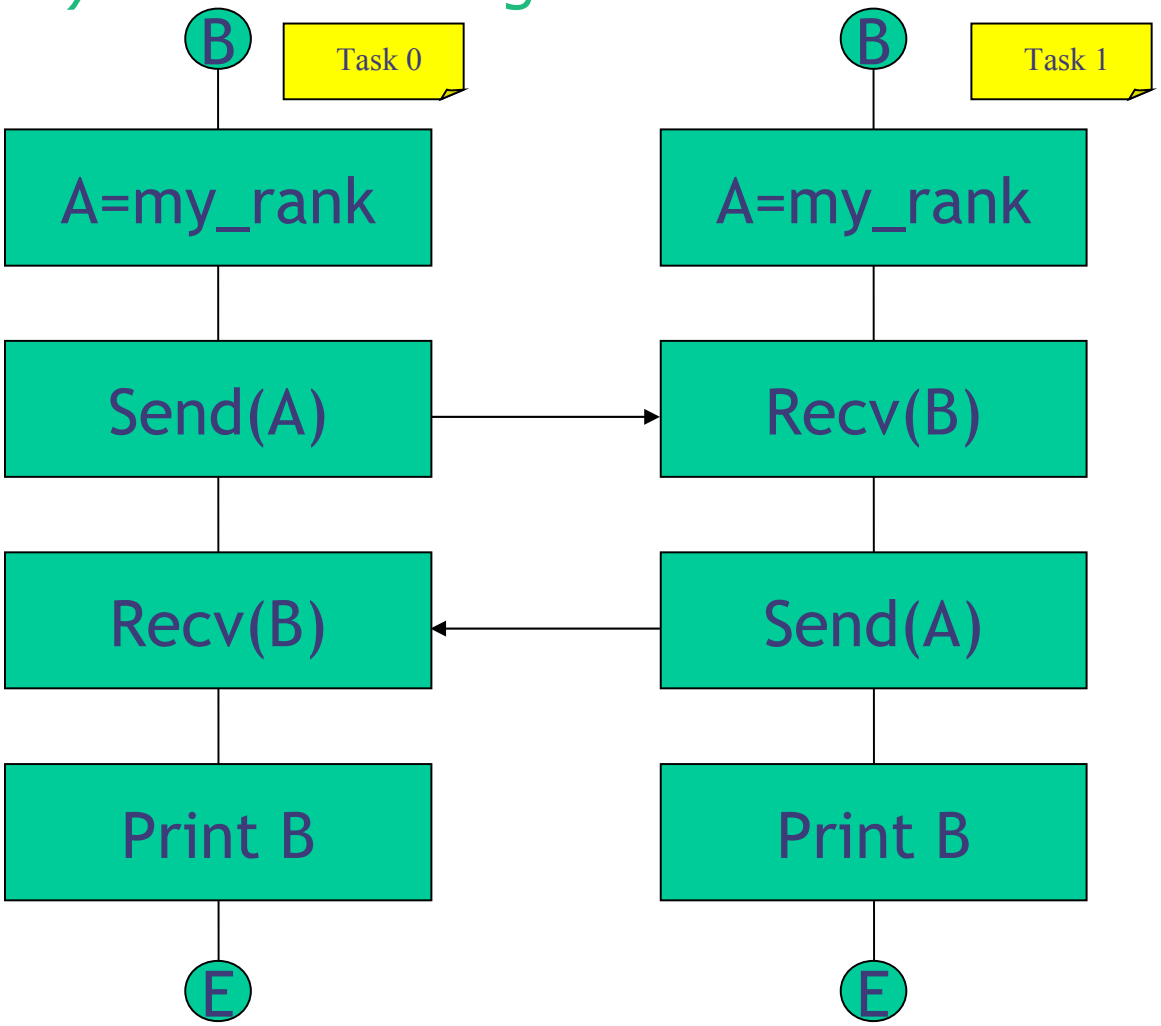
http://www.netlib.org/mpi/index.html

MPI Exercises

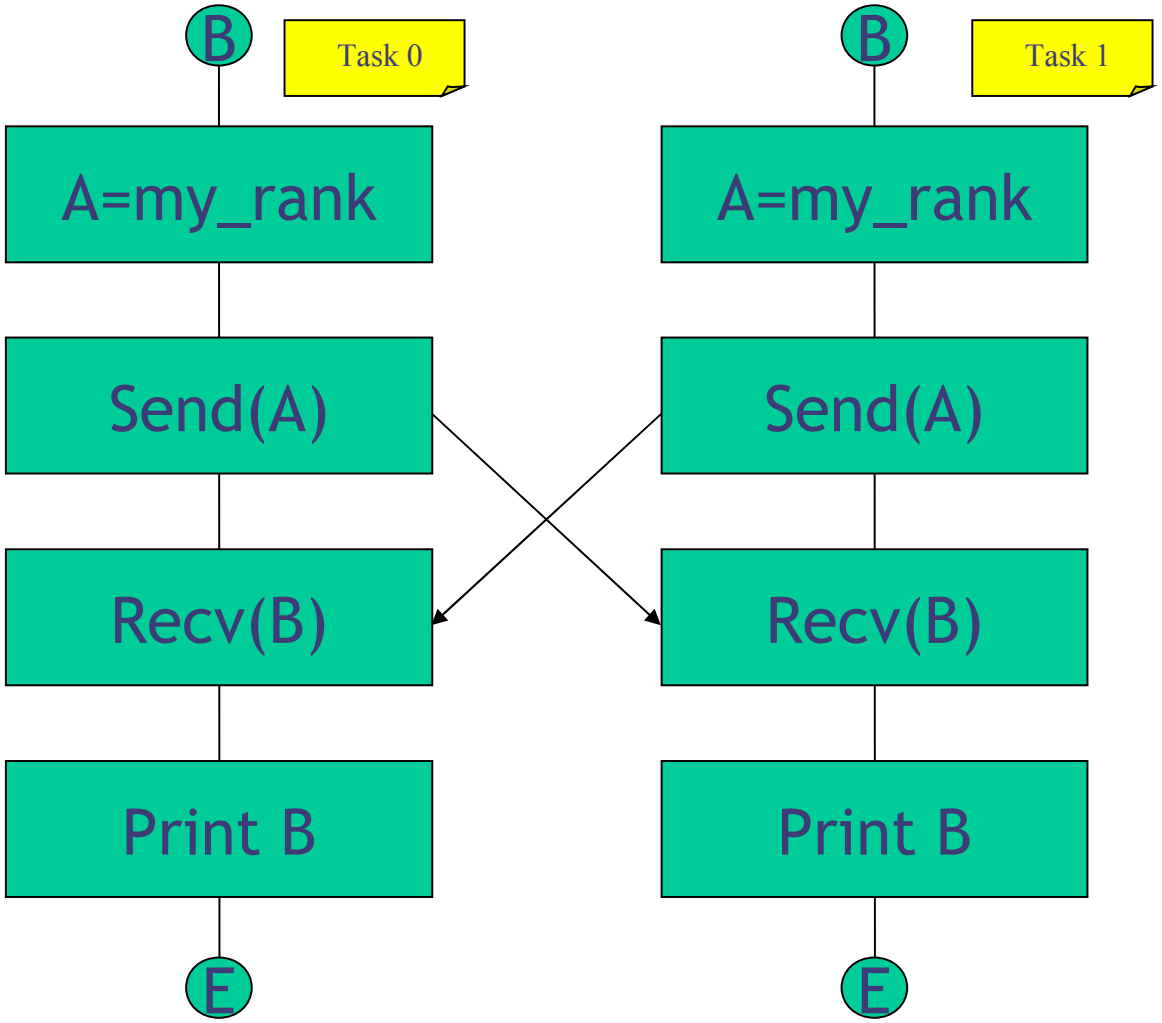
Using MPI, print:

1. Hello world
2. Hello world, I am proc X of total Y (from 1 to total 4 tasks)

a) Data exchange

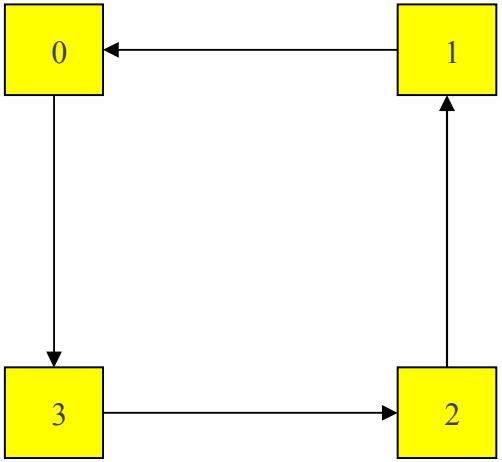


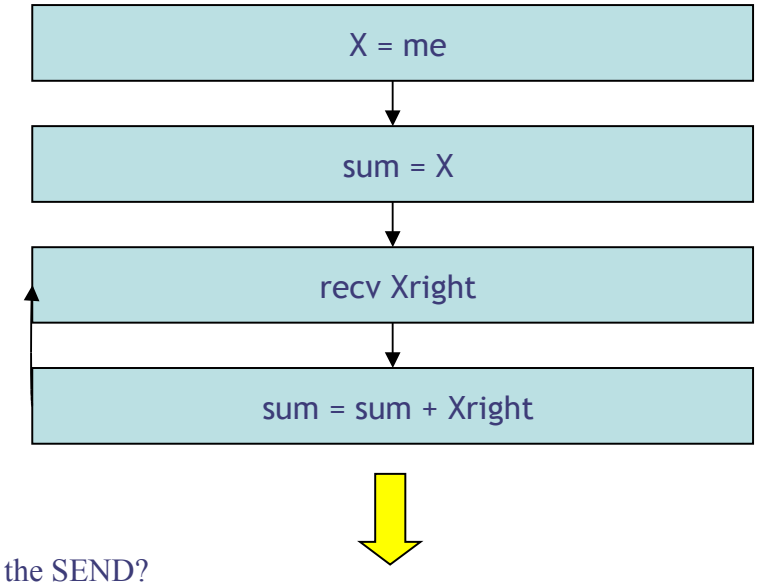
b) do it with deadlock and see it!



Sum with circular communications

Each process must print the sum of all ranks. Only communication with the left neighbour process is allowed, as showed in the figure.





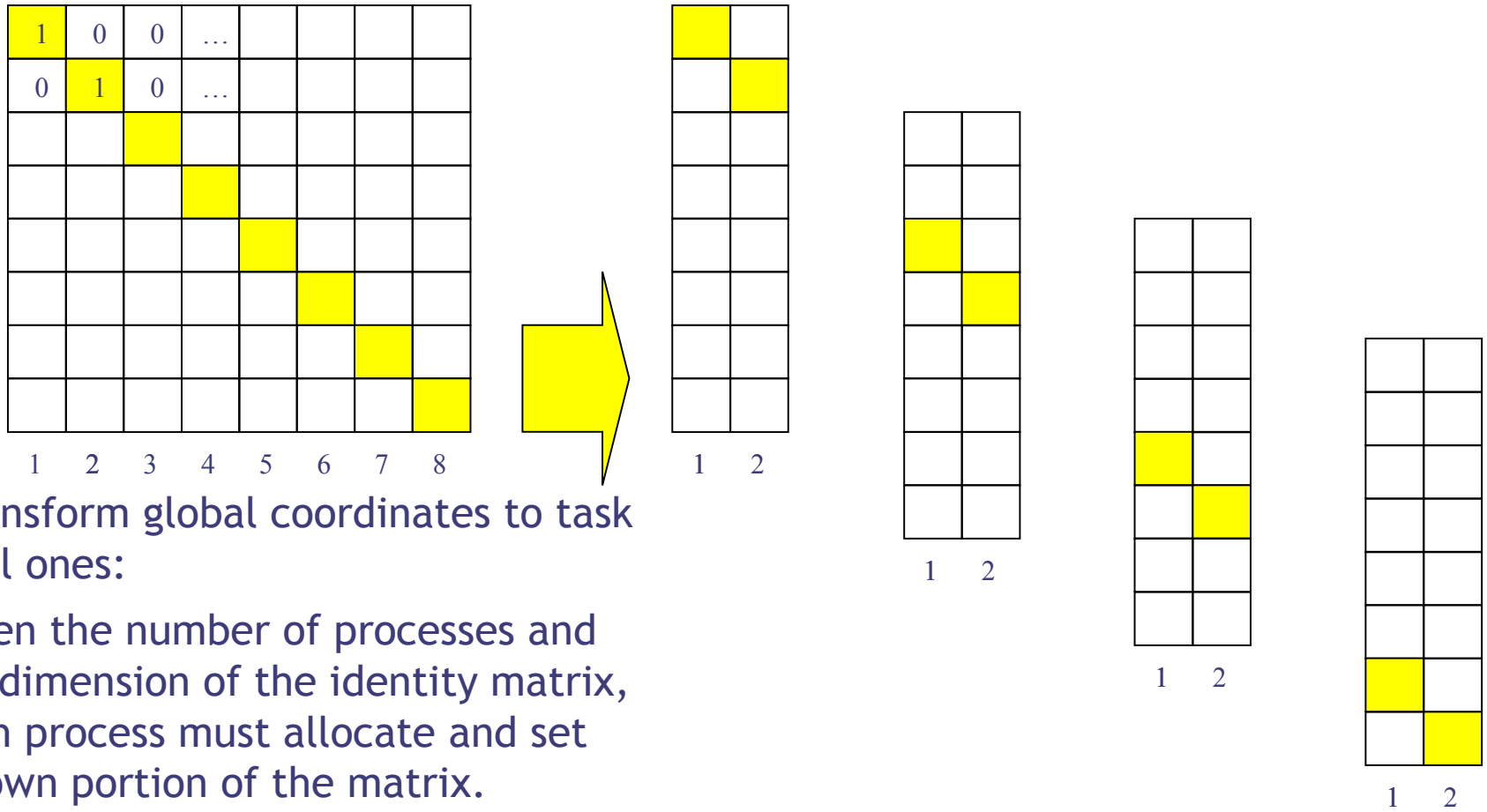
Who/when does the SEND?

What does it send?

How many times?

sum = 6 (on 4 processors)

Data distribution of identity



Transform global coordinates to task local ones:

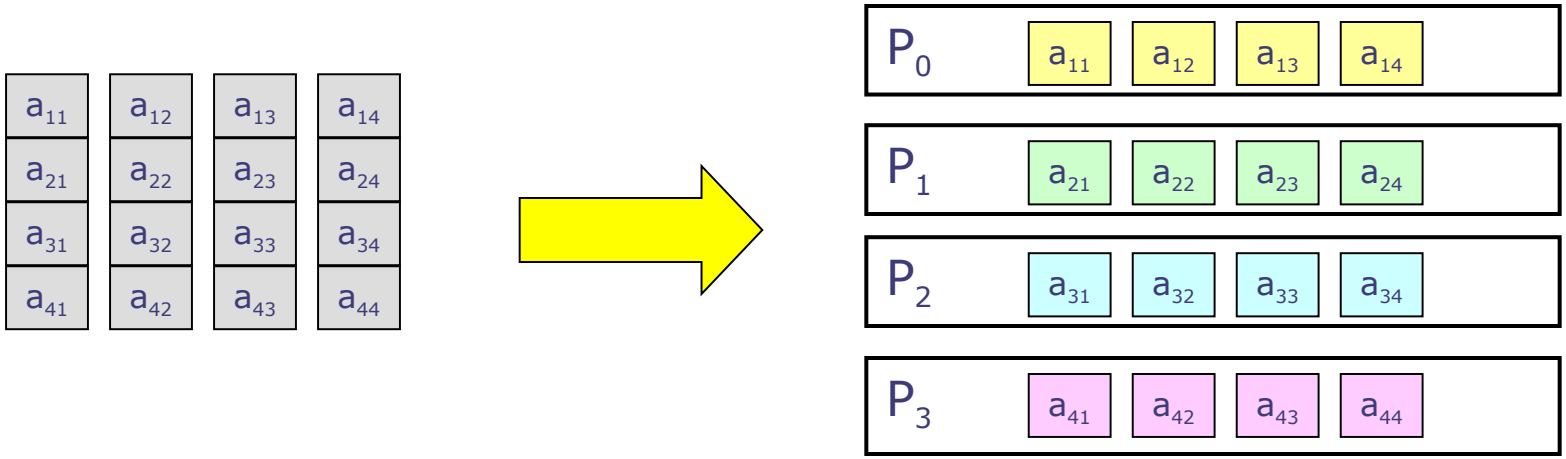
given the number of processes and the dimension of the identity matrix, each process must allocate and set its own portion of the matrix.

Parallel Matrix Multiplication

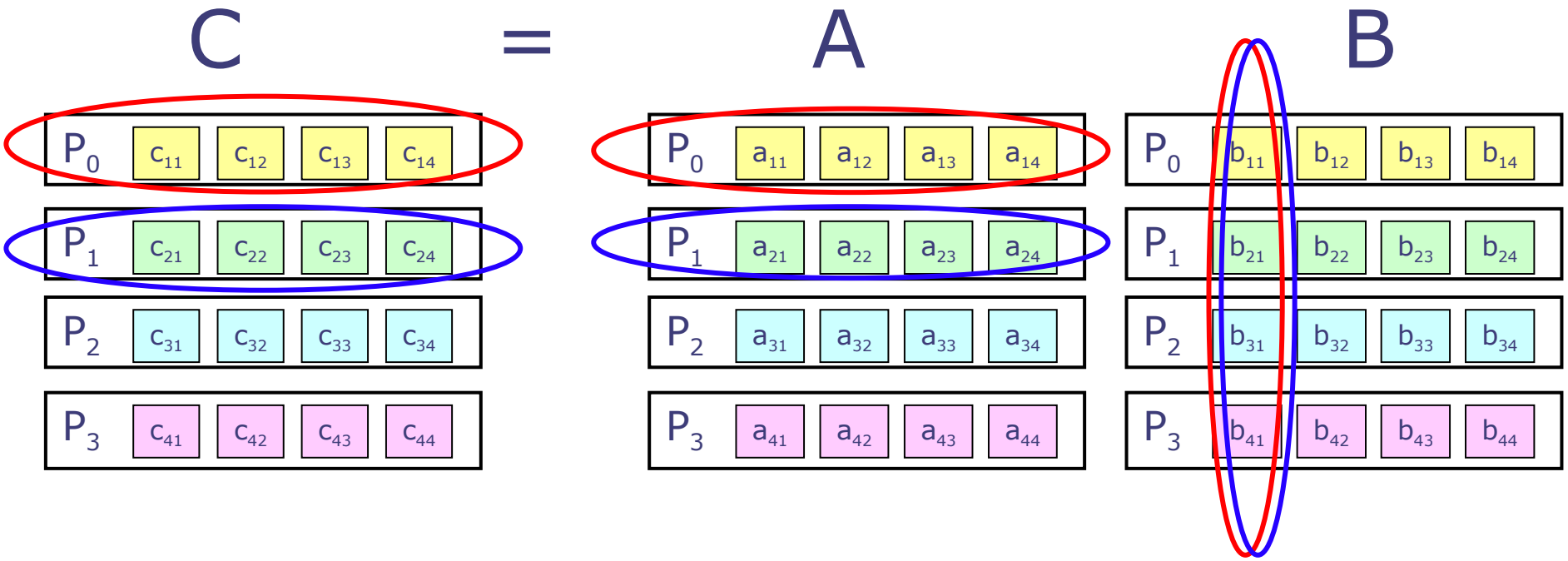
Write a **subroutine** implementing matrix multiplication and test it.

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

A, B and C being NxN matrices distributed by row among processes (at least 8x8).
Inizialize A and B matrices respectively as $a_{ij} = i*j$ and $b_{ij}=1/(i*j)$.
Try to minimize memory allocation and the number of MPI calls.



FORTRAN CASE: each process manage a row of elements (or blocks)



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

Each process compute the first element (block) of its own row

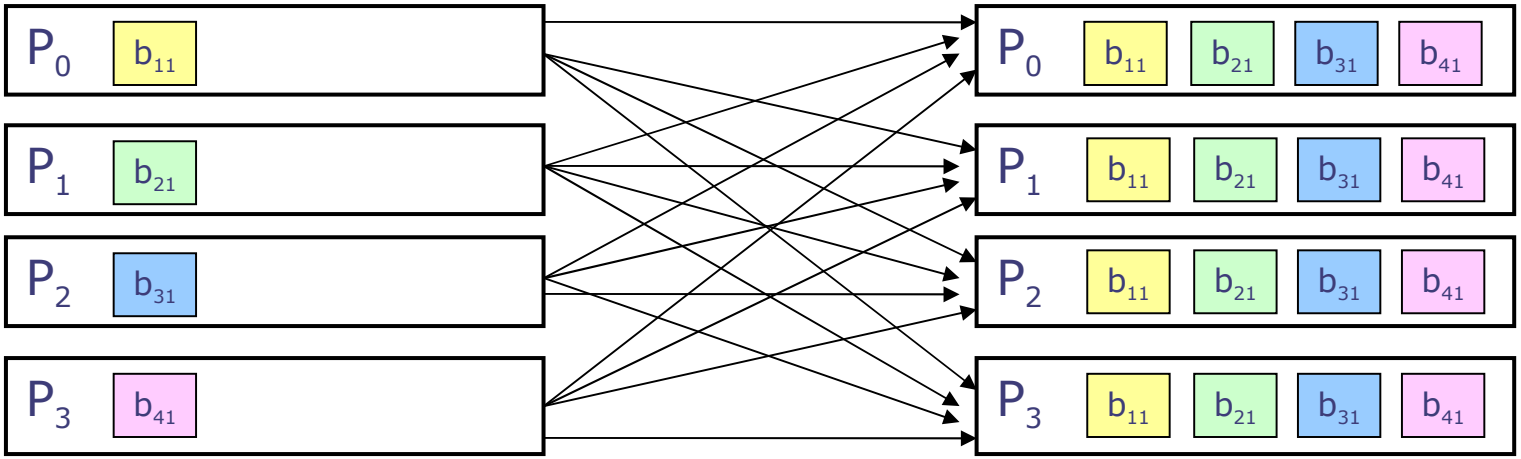
P_0	c_{11}	=	a_{11}	b_{11}	+	a_{12}	b_{21}	+	a_{13}	b_{31}	+	a_{14}	b_{41}
P_1	c_{21}	=	a_{21}	b_{11}	+	a_{22}	b_{21}	+	a_{23}	b_{31}	+	a_{24}	b_{41}
P_2	c_{31}	=	a_{31}	b_{11}	+	a_{32}	b_{21}	+	a_{33}	b_{31}	+	a_{34}	b_{41}
P_3	c_{41}	=	a_{41}	b_{11}	+	a_{42}	b_{21}	+	a_{43}	b_{31}	+	a_{44}	b_{41}

It's always the same data for all the tasks!

Allgather

Step 1: allgather

Perform an All gather, of the first colum of blocks



Step 2: local computation

Each processor calculate the first column 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}

Generalize

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