An Introduction to MPI Programming

Paul Burton

April 2015

Topics

- Introduction
- Initialising MPI & basic concepts
- Compiling and running a parallel program on the Cray
- Practical : “Hello World” MPI program
- Synchronisation
- Practical
- Data types and tags
- Basic sends and receives
- Practical
- Collective communications
- Reduction Operations
- MPI References

Introduction (1 of 4)

- Message Passing evolved in the late 1980's
- Cray was dominate in supercomputing
  - with very expensive shared-memory vector processors
- Many companies tried new approaches to HPC
- Workstation and PC Technology was spreading rapidly
- "The Attack of the Killer Micros"
- Message Passing was a way to link them together
  - many different flavours PVM, PARMACS, CHIMP, OCCAM
- Cray recognised the need to change
  - switched to MPP using cheap DEC Alpha microprocessors (T3D/T3E)
- But application developers needed portable software

Introduction (2 of 4)

- **Message Passing Interface (MPI)**
  - The MPI Forum was a combination of end users and vendors (1992)
  - defined a standard set of library calls in 1994
  - Portable across different computer platforms
  - Fortran and C Interfaces
- **Used by multiple tasks to send and receive data**
  - Working together to solve a problem
  - Data is decomposed (split) into multiple parts
  - Each task handles a separate part on its own processor
  - Message passing to resolve data dependencies
- **Works within a node and across Distributed Memory Nodes**
- **Can scale to thousands of processors**
  - Subject to constraints of Amdahl's Law

Introduction (3 of 4)

- **The MPI standard is large**
  - **Well over 100 routines in MPI version 1**
  - **Result of trying to cater for many different flavours of message passing and a diverse range of computer architectures**
  - **And an additional 100+ in MPI version 2 (1997)**
- **Many sophisticated features**
  - **Designed for both homogenous and heterogeneous environments**
- **But most people only use a small subset**
  - **IFS was initially parallelised using Parmacs**
  - **This was replaced by about 10 MPI routines**
    - **Hidden within “MPL” library**

Introduction (4 of 4)

- **This course will look at just a few basic routines**
  - **Fortran Interface Only**
  - **MPI version 1.2**
  - **SPMD (Single Program Multiple Data)**
  - **As used at ECMWF in IFS**
- **A mass of useful material on the Web**
  - **Google is your friend!**

SPMD

- **The SPMD model is by far the most common**
  - **Single Program Multiple Data**
  - **One program executes multiple times simultaneously**
  - **The problem is divided across the multiple copies**
  - **Each work on a subset of the data**
- **MPMD**
  - **Multi Program Multiple Data**
  - **Different executable on different processors**
  - **Useful for coupled models for example**
  - **Part of the MPI 2 standard**
  - **Not currently used by IFS**
  - **Can be mimicked in SPMD mode**
    - **Top level branch deciding which “program” (subroutine) this task will run**

Some definitions

- **Task**
  - one running instance (copy) of a program
  - Equivalent to a UNIX process
  - Basic unit of an MPI parallel execution
  - May run on one processor
    - Or across many if OpenMP is used as well
    - Or many tasks on one processor (not a good idea!)
- **Master**
  - the master task is the first task in a parallel program : TaskID=0
- **Slave**
  - all other tasks in a parallel program
  - Nothing intrinsically different between master/slave – but the parallel program may treat them differently

The simplest MPI program..........

- **Lets start with "hello world"**
- **Introduces**
  - **4 essential housekeeping routines**
  - **the "`use mpi`" statement**
  - **the concept of Communicators**

```
program hello

implicit none

print *,"Hello world"

end
```

Hello World with MPI

```
program hello

implicit none
use mpi
integer:: ierror,ntasks,mytask

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)

print *,"Hello world from task ",mytask," of ",ntasks

call MPI_FINALIZE(ierror)

end
```

use mpi : The MPI header file

```
use mpi
```

- **The MPI header file**
- **Always include in any routine calling an MPI function**
- **Contains declarations for constants used by MPI**
- **May contain interface blocks, so compiler will tell you if you make an obvious error in arguments to MPI library**
  - **This is not mandated by the standard so you shouldn't rely on it. You may want to test Cray's mpi to see if it does!**
- **In Fortran77 use `include 'mpif.h'` instead**

Hello World with MPI

```
program hello

implicit none
use mpi
integer:: ierror,ntasks,mytask

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)

print *,"Hello world from task ",mytask," of ",ntasks

call MPI_FINALIZE(ierror)

end
```

MPI_INIT

```
integer:: ierror
call MPI_INIT(ierror)
```

- **Initializes the MPI environment**
- **Expect a return code of zero for ierror**
  - **If an error occurs the MPI layer will normally abort the job**
  - **best practise would check for non zero codes**
  - **we will ignore for clarity – but see later slides for `MPI_ABORT`**
- **On the Cray all tasks execute the code before `MPI_INIT`**
  - **this is an implementation dependent feature**
  - **try not to do anything that alters the state of the system before this, eg. I/O**

Hello World with MPI

```
program hello

implicit none
use mpi
integer:: ierror,ntasks,mytask

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)

print *,"Hello world from task ",mytask," of ",ntasks

call MPI_FINALIZE(ierror)

end
```

MPI_COMM_WORLD

- **An MPI communicator**
- **Constant integer value from "`use mpi`"**
- **Communicators define sets or groups of tasks**
  - **dividing programs into subsets of tasks often not necessary**
  - **IFS also creates and uses some additional communicators**
    - **useful when doing collective communications**
    - **Useful if you want to dedicate a subset of tasks to a special job (eg. I/O server)**
  - **advanced topic**
- **`MPI_COMM_WORLD` means all tasks**
  - **many MPI programs only use `MPI_COMM_WORLD`**
  - **All our examples only use `MPI_COMM_WORLD`**

Hello World with MPI

```
program hello

implicit none
use mpi
integer:: ierror,ntasks,mytask

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)

print *,"Hello world from task ",mytask," of ",ntasks

call MPI_FINALIZE(ierror)

end
```

MPI_COMM_SIZE

```
integer:: ierror,ntasks
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
```

- **Returns the number of parallel tasks in the variable “ntasks”**
  - **the number of tasks is defined from the aprun command which starts the parallel executable**
- **Value can be used to help decompose the problem**
  - **in conjunction with Fortran allocatable/automatic arrays**
  - **avoid the need to recompile for different processor numbers**

Hello World with MPI

```
program hello

implicit none
use mpi
integer:: ierror,ntasks,mytask

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)

print *,"Hello world from task ",mytask," of ",ntasks

call MPI_FINALIZE(ierror)

end
```

MPI_COMM_RANK

```
integer:: ierror, mytask
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)
```

- **Returns the rank of the task in `mytask`**
  - **In the range `0` to `ntasks-1`**
    - **Easy to make mistakes with this as Fortran arrays normally run `1:n`**
  - **Used as a task identifier when sending/receiving messages**

Hello World with MPI

```
program hello

implicit none
use mpi
integer:: ierror,ntasks,mytask

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)

print *,"Hello world from task ",mytask," of ",ntasks

call MPI_FINALIZE(ierror)

end
```

MPI_FINALIZE

```
integer:: ierror
call MPI_FINALIZE(ierror)
```

- **Tell the MPI layer that we have finished**
- **Any MPI call after this is an error**
  - **Like MPI_INIT, the MPI standard does not mandate what happens after an MPI_FINALIZE – cannot guarantee that all tasks still execute after this point**
- **Does not stop the program – at least one (probably all!) tasks will continue to run**

MPI_ABORT

```
integer:: ierror
call MPI_ABORT(MPI_COMM_WORLD,ierror)
```

- **Causes all tasks to abort**
- **Even if only one task makes call**

PBSPro and MPI

- **Many varied ways of defining your requirements**
- **For the exercises we'll keep it as simple as possible**
  - **Create an interactive shell in which you can run parallel jobs in up to one node (48 hyperthreaded CPUs)**
  - **You won't need to wait every time you run an executable!**
  - **Don't forget to log out when you're finished!**
  - **Not recommended for regular use!**

```
$ ssh cca  # or ccb
```

queue "np" — one node

```
$ qsub -q np -I -l EC_nodes=1 -l EC_hyperthreads=2
```

interactive — Use hyperthreading

Compiling an MPI Program

- **Very easy using modules**
  - **Automatically adds all the flags/libraries required for MPI**

```
$ module load PrgEnv-cray     # Use Cray compilers
```

or

```
$ module load PrgEnv-intel    # Use Intel compilers
```

or

```
$ module load PrgEnv-gnu      # Use Gnu compilers
----

$ ftn hello.f90               # produces a.out
```

or

```
$ ftn -c hello.f90            # produces hello.o
```

and

```
$ ftn hello.o -o hello.exe    # produces hello.exe
```

Running an MPI Program

- **aprun**
  - **Details and many options covered in other lectures**
  - **Here we will use a very simple form**
  - **Run from the MOM node, launches the parallel executable on the parallel (ESM) node(s)**

```
$ aprun -n 4 <executable>
```

First Practical

- **Copy all the practical exercises to your account on cca or ccb:**
  - `cd $HOME`
  - `mkdir mpi_course ; cd mpi_course`
  - `cp -r ~trx/mpi.2015/* .`

- **Exercise1a**
  - **Run your own Hello World program with MPI**

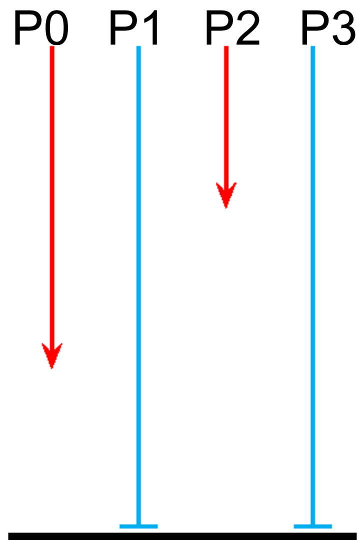
- **See the README for details**

MPI_BARRIER

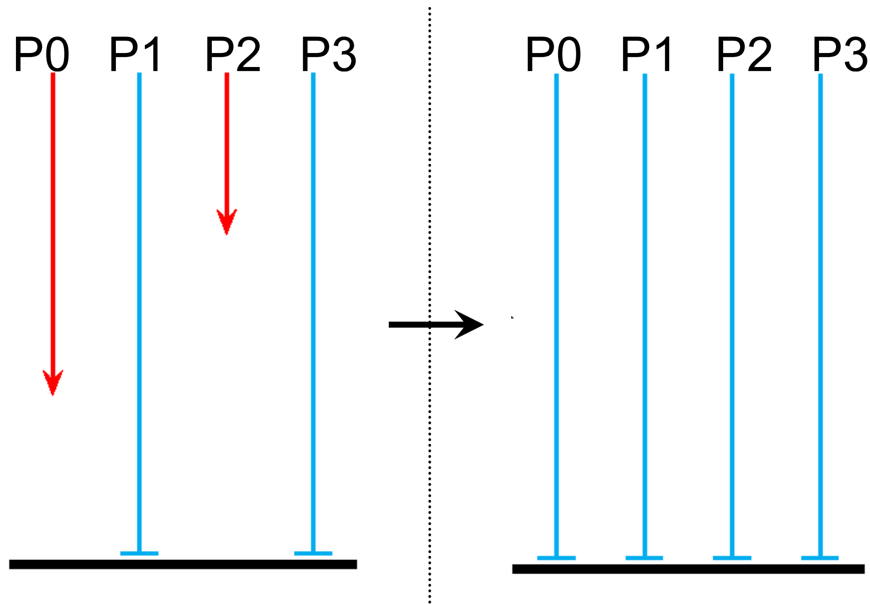
```
integer:: ierror
call MPI_BARRIER(MPI_COMM_WORLD,ierror)
```

- **Forces all tasks (in a communicator group) to synchronise**
  - **for timing points**
  - **to improve output of prints**
    - **can be used to force ordering of events**
  - **to separate different communications phases**
- **A task waits in the barrier until all tasks reach it**
- **Then every task completes the call together**
- **Deadlock if one task does not reach the barrier**
  - **`MPI_BARRIER` will wait until the task reaches its cpu limit**

MPI_BARRIER

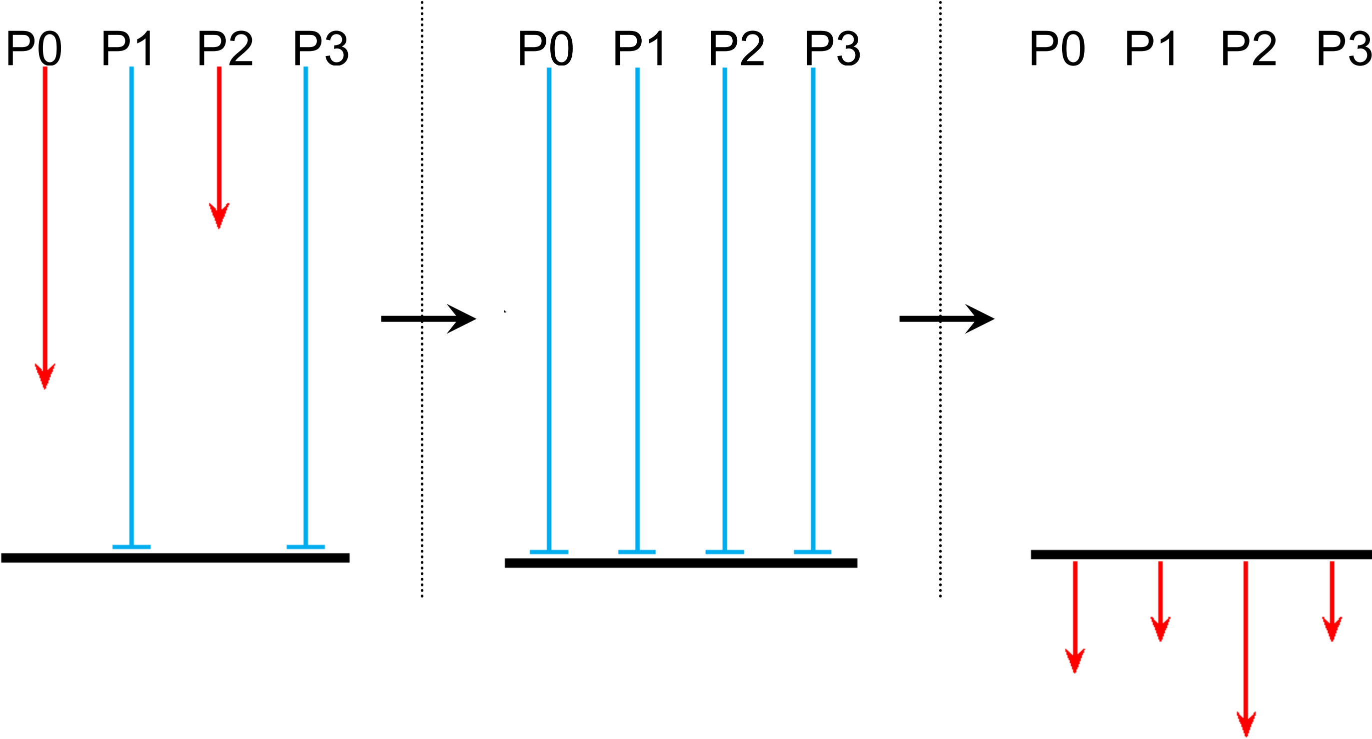


MPI_BARRIER



IDRIS-CNRS

MPI_BARRIER



IDRIS-CNRS

Second Practical

- **Forcing the ordering of output**
- **Exercise 1b – see the README file for more details…**

Basic Sends and Receives

- `MPI_SEND`
  - sends a message from one task to another
- `MPI_RECV`
  - receives a message from another task
- **A message is just data with some form of identification**
  - think of it as an email – some information and some headers
    - To: Where the message should be sent to
    - Subject: Some description of the contents (in MPI, a "tag")
    - Body: The data itself (can be any size), various Fortran types
- **You program the logic to send and receive messages**
  - the sender and receiver are working together
  - every send must have a corresponding receive

MPI Datatypes

- **MPI can send variables of any Fortran type**
  - `integer, real, real*8, logical`, ...
  - it needs to know the type
- **There are predefined constants used to identify types**
  - `MPI_INTEGER, MPI_REAL, MPI_REAL8, MPI_LOGICAL`......
  - Defined by "`use mpi`"
- **Also user defined data types**
  - MPI allows you create types created out of basic Fortran types (rather like a Fortran 90 structure)
  - Allows strided (non contiguous) data to be communicated
  - advanced topic

MPI Tags

- **All messages are given an integer TAG value**
  - **standard says maximum value is at least 32768 (2^31)**
  - ```
    CALL MPI_Comm_get_attr (MPI_COMM_WORLD,MPI_TAG_UB,
                            maxtag, flag, error)
    ```
- **This helps to identify a message (like an email's "subject")**
- **Particularly useful when sending multiple messages**
- **You decide what tag values to use**
  - **Good ideas to use separate ranges of tags eg:**
    - **1000, 1001, 1002..... in routine a**
    - **2000, 2001, 2002.... in routine b**
    ```

MPI_SEND

```
FORTRAN_TYPE:: sbuf
integer:: count, dest, tag, ierror
call MPI_SEND( sbuf, count, MPI_TYPE, dest, tag, &
               MPI_COMM_WORLD, ierror)
```

SBUF	**the array being sent**	**input**
COUNT	**the number of elements to send**	**input**
MPI_TYPE	**type of SBUF eg MPI_REAL**	**input**
DEST	**the task id of the receiver**	**input**
TAG	**the message identifier**	**input**

MPI_RECV

```
FORTRAN_TYPE:: rbuf
integer:: count, source, tag, status(MPI_STATUS_SIZE),ierror
call MPI_RECV( rbuf, count, MPI_TYPE, source, tag, &
               MPI_COMM_WORLD, status, ierror)
```

- **RBUF** — **the array being received** — **output**
- **COUNT** — **the length of RBUF** — **input**
- ***MPI_TYPE*** — **type of RBUF eg MPI_REAL** — **input**
- **SOURCE** — **the task id of the sender** — **input**
- **TAG** — **the message identifier** — **input**
- **STATUS** — **information about the message** — **output**

More on MPI_RECV

- **`MPI_RECV` will block (wait) until the message arrives**
  - **if message never sent then deadlock**
    - **task will wait until it reaches cpu time limit, and then dies**
- **Order in which messages are received**
  - **For a given pair of processors using the same communicator, the MPI standard guarantees the messages will be received in the same order they were sent**
- **This means you need to be careful**
  - **If you are receiving multiple messages from the same task, you MUST do the `MPI_RECV`s in the same order as the `MPI_SEND`s**
  - **Otherwise the first `MPI_RECV` will wait forever, and eventually die**
  - ***What happens if you don't know the ordering of the `MPI_SEND`s?***

How to be less specific on MPI_RECV

- **The source and tag can be more open**
  - **`MPI_ANY_SOURCE` means receive from any sender**
  - **`MPI_ANY_TAG` means receive any tag**
  - **Useful in more complex communication patterns**
  - **Used to receive messages in a more random order**
  - **helps smooth out load imbalance**
  - **May require over-allocation of receive buffer**
- **But how do we know what message we've received?**
  - **`status(MPI_SOURCE)` will contain the actual sender**
  - **`status(MPI_TAG)` will contain the actual tag**

A simple example

```
subroutine transfer(values,len,mytask)
implicit none
use mpi
integer:: mytask,len,source,dest,tag,ierror,status(MPI_STATUS_SIZE)
real::    values(len)
tag = 12345
if(mytask.eq.0) then
   dest = 1
   call MPI_SEND(values,len,MPI_REAL,dest,tag,MPI_COMM_WORLD,ierror)
elseif(mytask.eq.1) then
   source = 0
   call MPI_RECV(values,len,MPI_REAL,source,tag,MPI_COMM_WORLD,status,ierror)
endif
end
```

Third Practical

- **Sending and receiving a message**
- **Exercise 1c – see the README file for more details…**

Collective Communications (1)

- **SEND/RECV is pairwise communication**
- **Often we want to do more complex communication patterns**
- **For example**
  - **Send the same message from one task to many other tasks**
  - **Receive messages from many tasks onto many other tasks**
- **We could write this with `MPI_SEND` & `MPI_RECV`**
  - **How?**
  - **Why not?**

Collective Communications

- **MPI contains Collective Communications routines**
  - **called by all tasks (in a comminicator group) together**
  - **replace multiple send/recv calls**
  - **easier to code and understand**
  - **can be more efficient**
  - **the MPI library may optimise the data transfers**
- **We will look at `MPI_BCAST` and `MPI_GATHER`**
- **Other routines will be summarised**
- **The diagrams are schematic**
  - **Help to conceptualise the data movement**
  - **The MPI library and machine hardware may actually be doing a more complex (and hopefully efficient!) communication pattern**
- **IFS uses a few collective routines, sometimes we hand code our own**

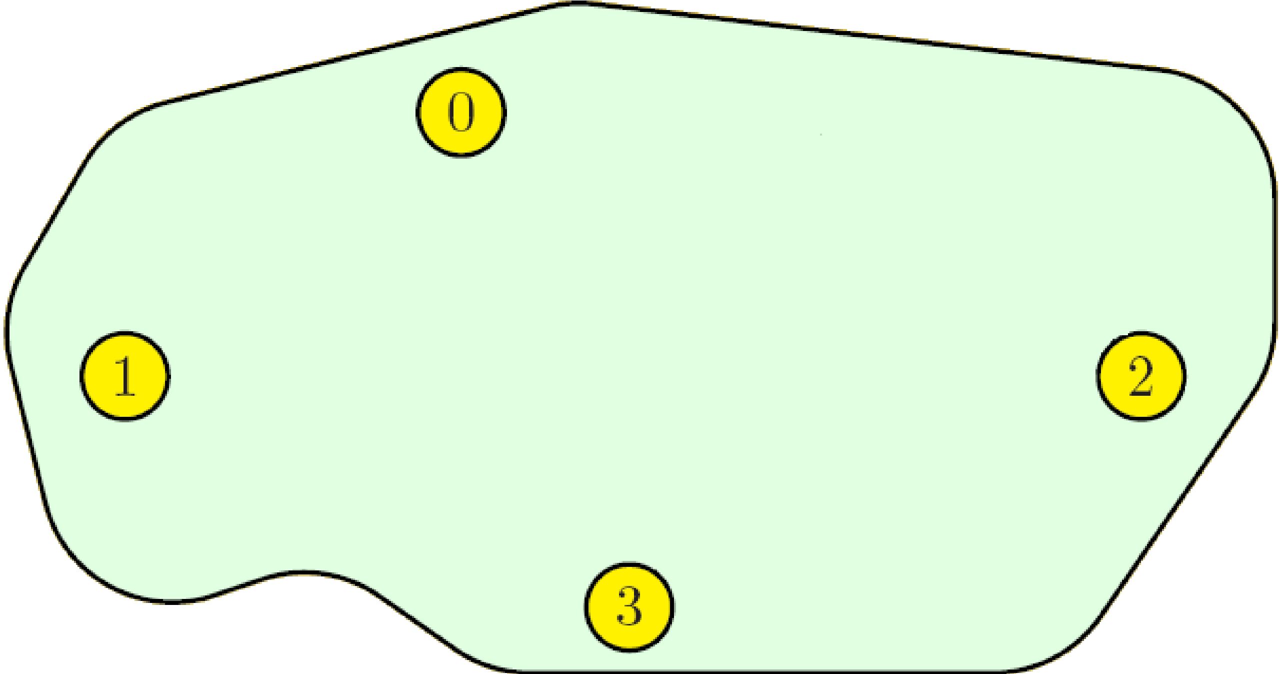
MPI_BCAST

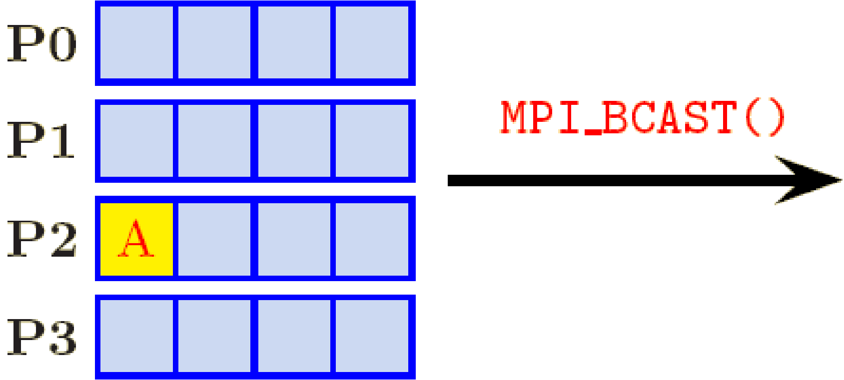
```
FORTRAN_TYPE:: buff
integer:: count, root, ierror
call MPI_BCAST( buff,count,MPI_TYPE,root,MPI_COMM_WORLD,ierror)
```

• **ROOT**	**task doing broadcast**	**input**
• **BUFF**	**array being broadcast**	**input/output**
• **COUNT**	**the number of elements**	**input**
• ***MPI_TYPE***	**the kind of variable**	**input**

The contents of `buff` are sent from task id `root` to all other tasks. Equivalent to putting `MPI_SEND` in a loop and matching `MPI_RECV`s

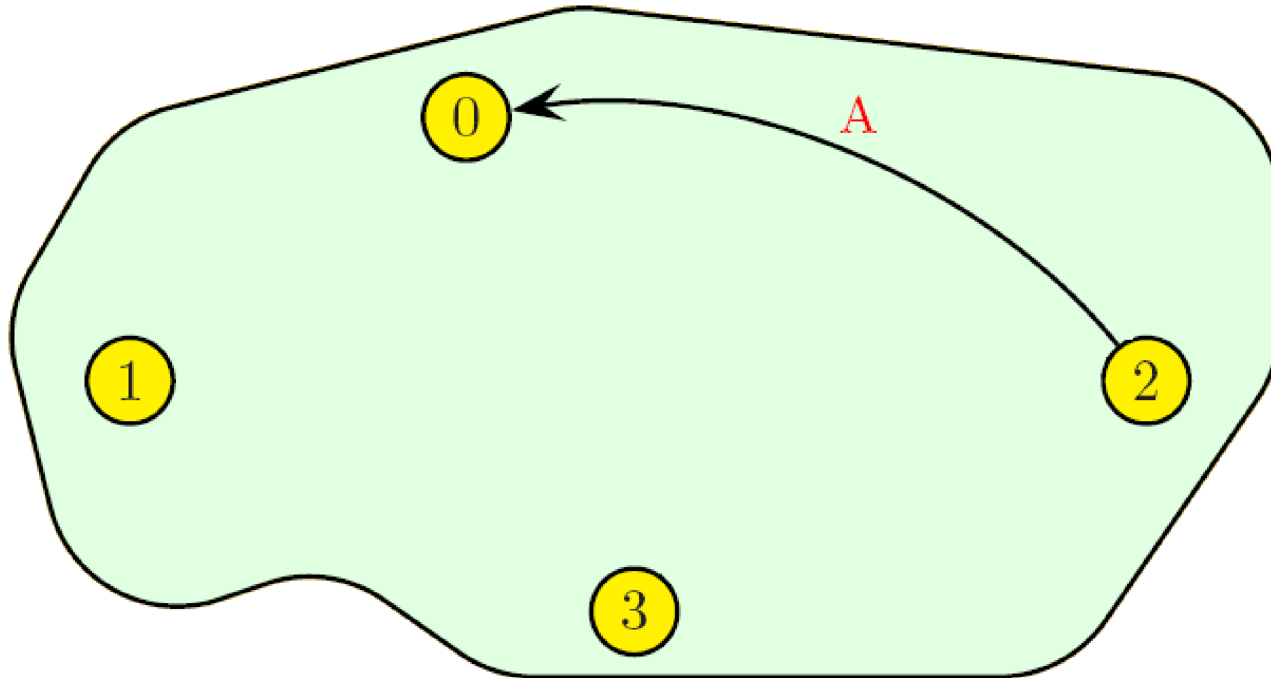
MPI_BCAST

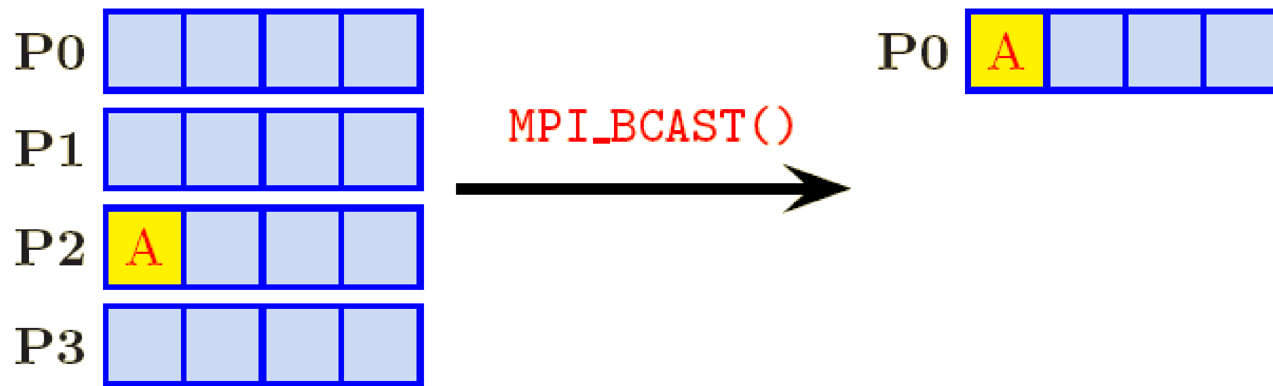




IDRIS-CNRS

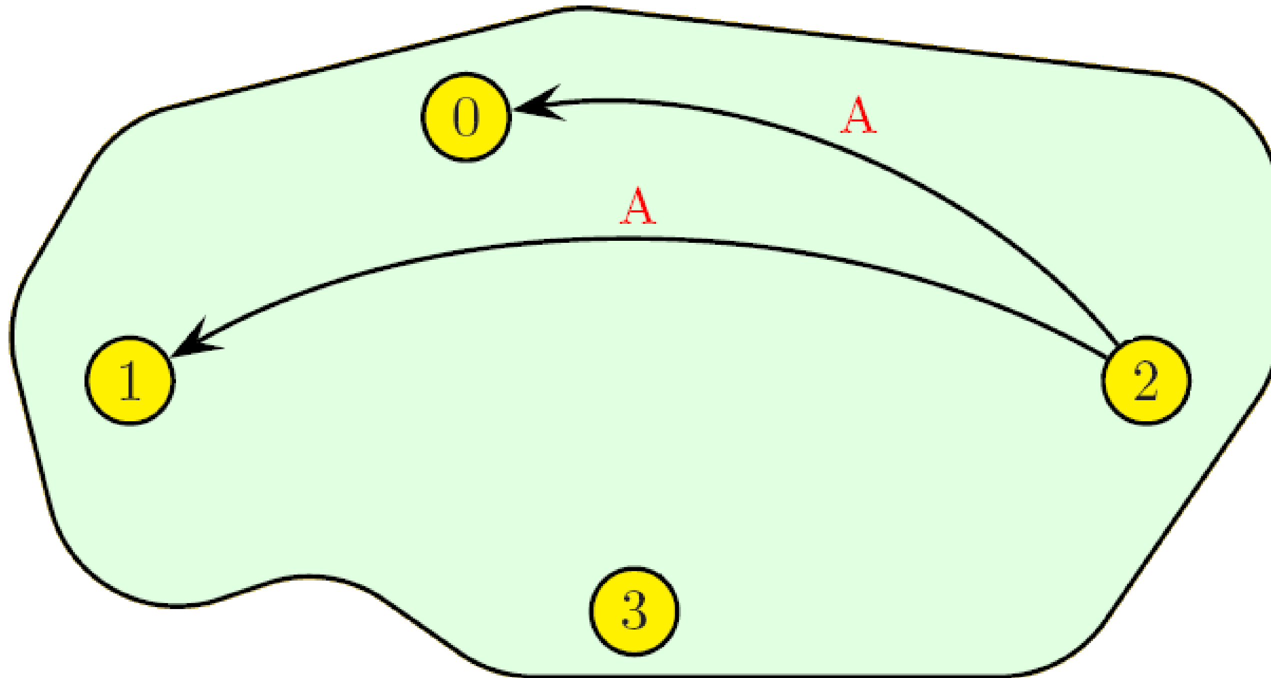
MPI_BCAST

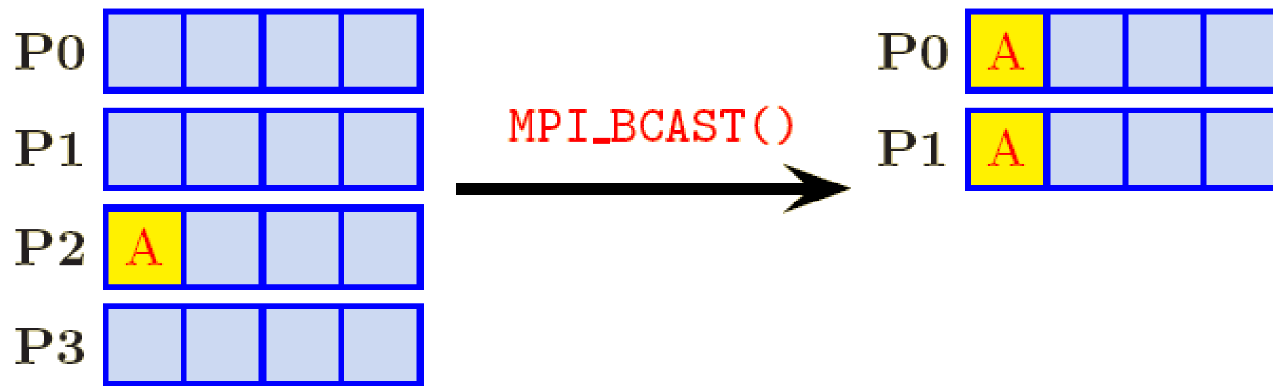




IDRIS-CNRS

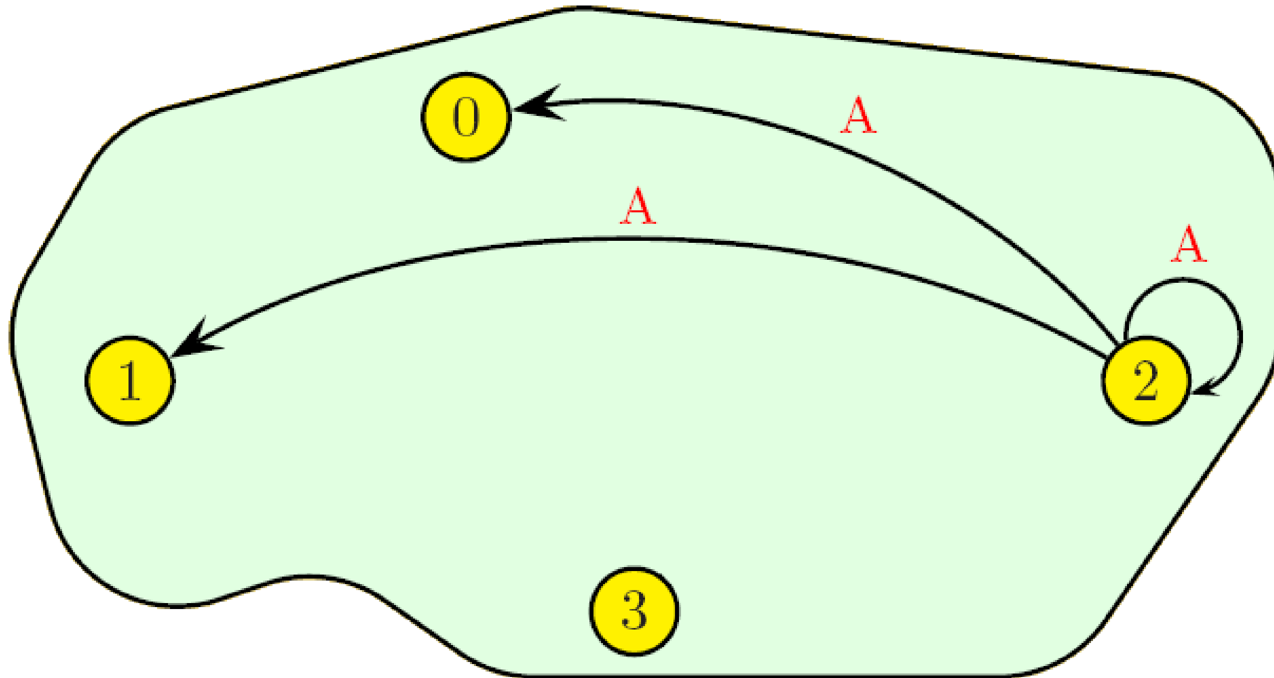
MPI_BCAST

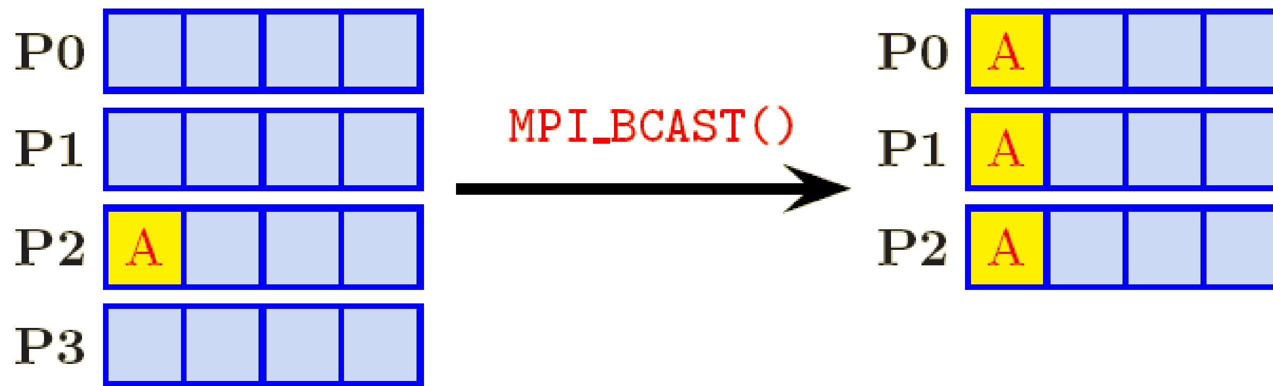




IDRIS-CNRS

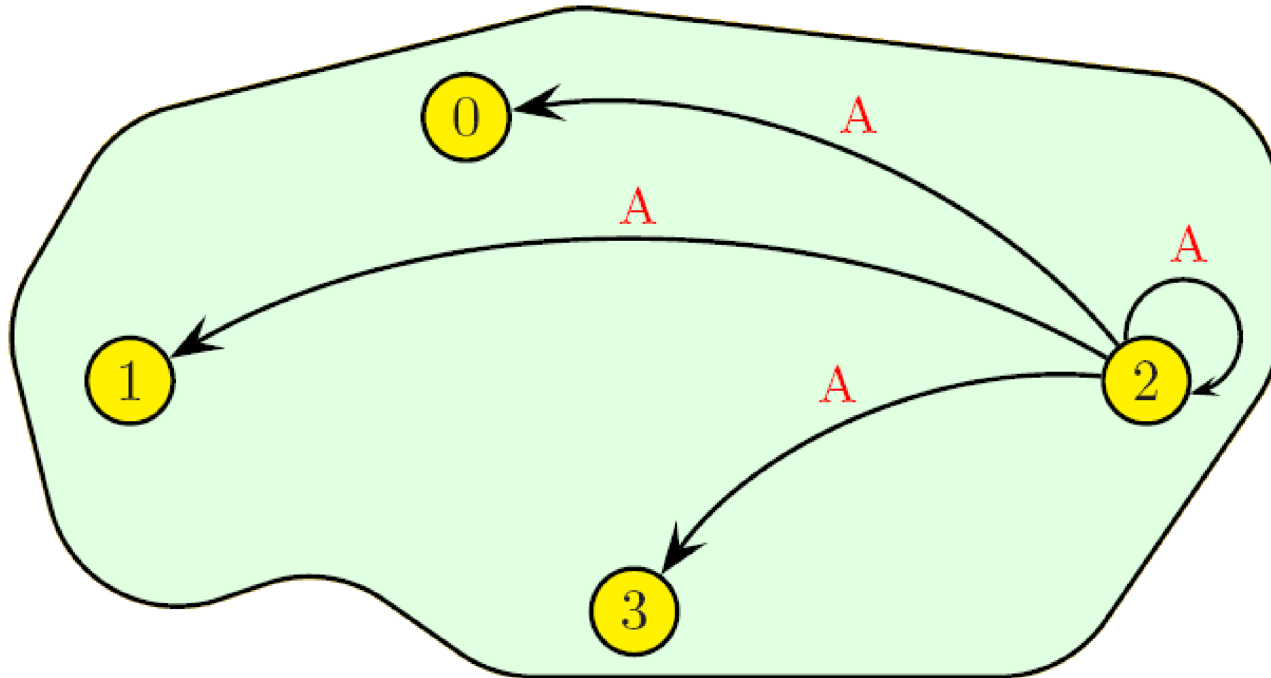
MPI_BCAST

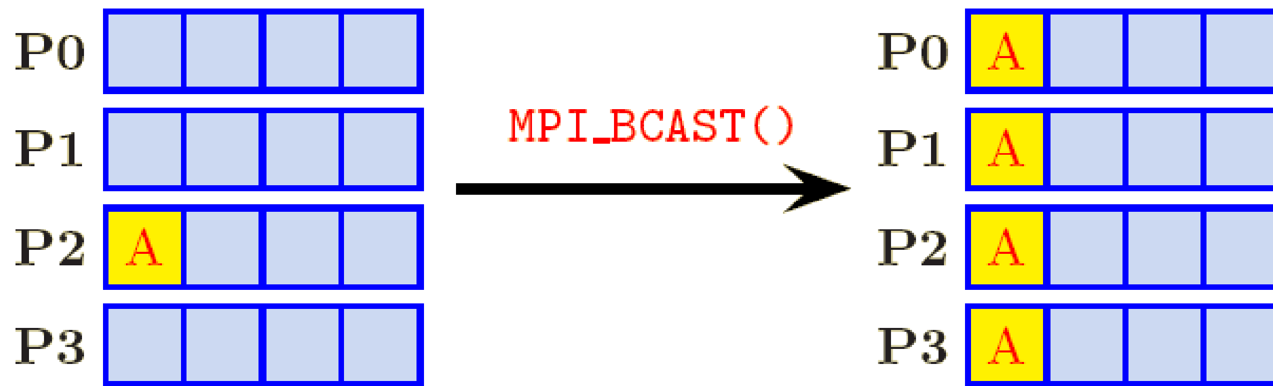




IDRIS-CNRS

MPI_BCAST





IDRIS-CNRS

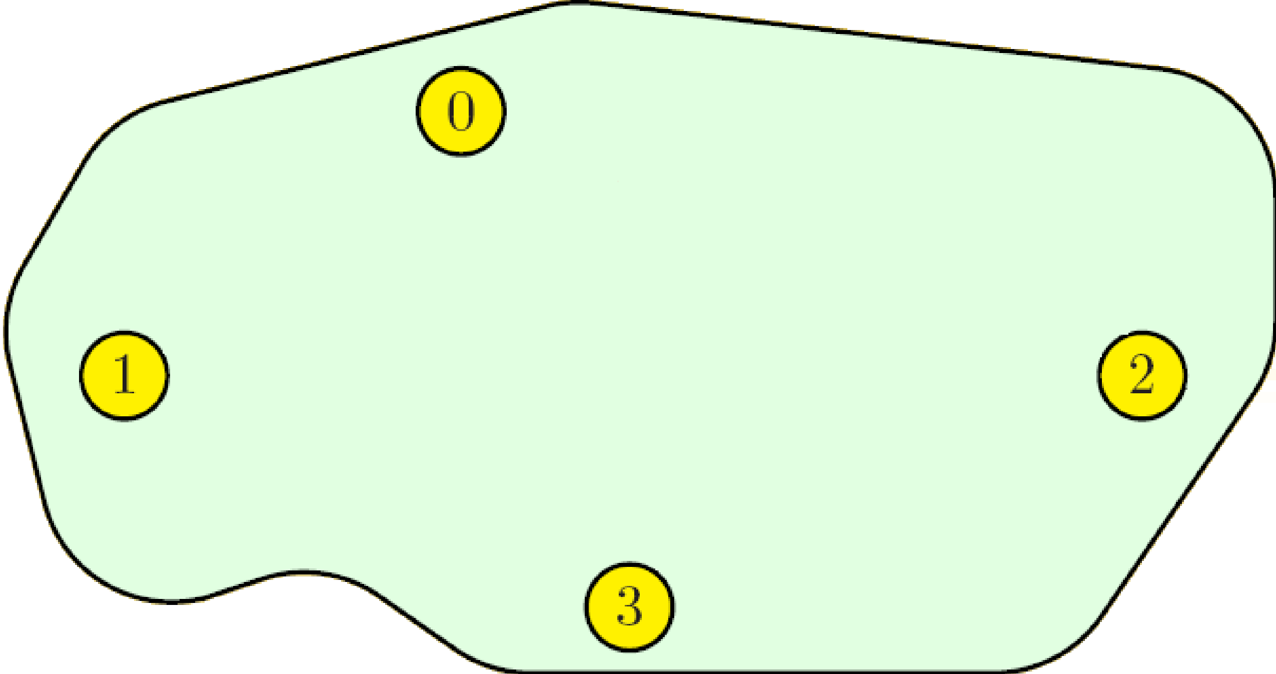
MPI_GATHER

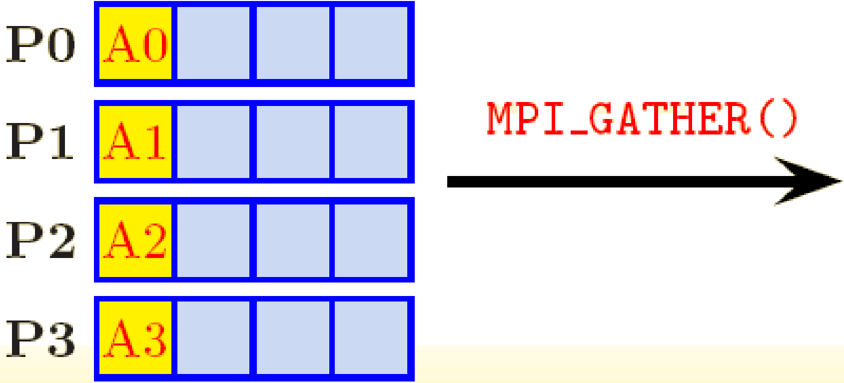
```
FORTRAN_TYPE:: sbuff, rbuff
integer:: count, root, ierror
call MPI_GATHER( sbuff,scount,MPI_TYPE, &
                 rbuff,rcount,MPI_TYPE,root,MPI_COMM_WORLD,ierror)
```

● **ROOT**	**task doing gather**	**input**
● **SBUFF**	**array being sent**	**input**
● **RBUFF**	**array being received**	**output**
● **[S/R]COUNT**	**number of items to/from each task**	**input**

The contents of `sbuff` are sent from every task to task id `root` and received (concatenated in rank order) in array `rbuff`. Could also be done by putting `MPI_RECV` in a loop.

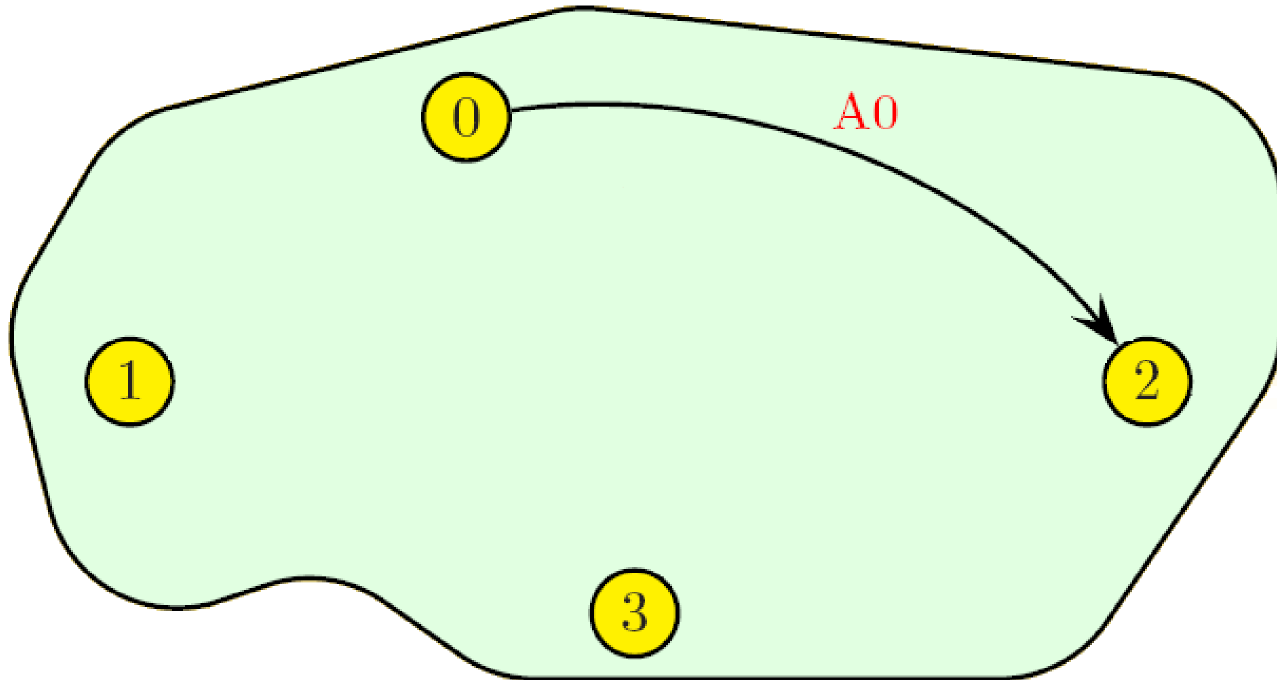
MPI_GATHER

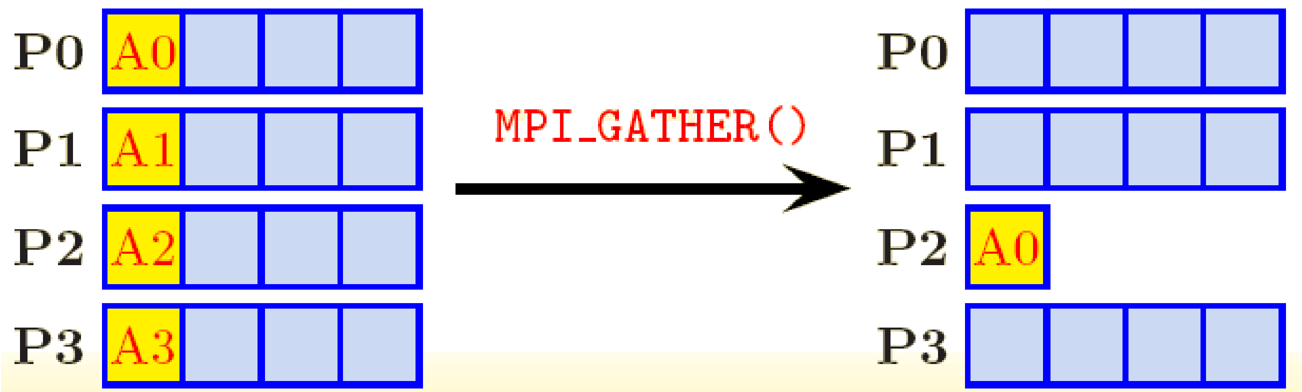




IDRIS-CNRS

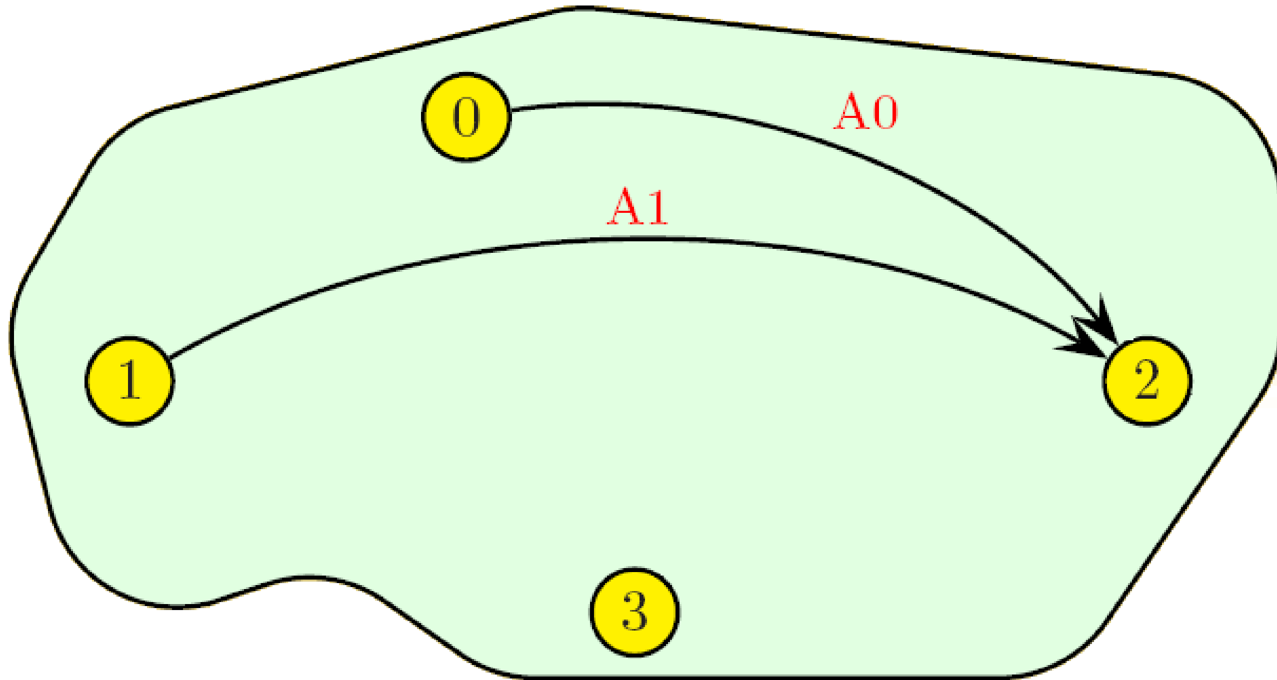
MPI_GATHER

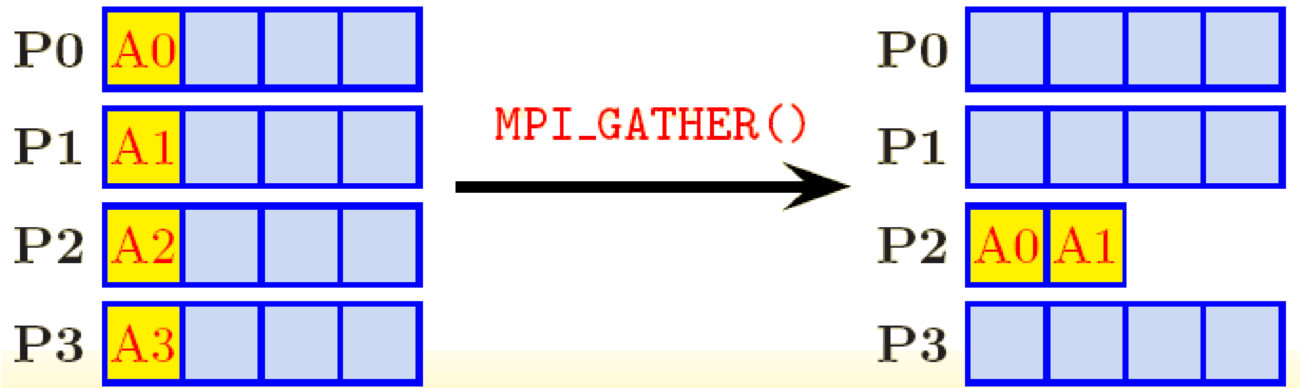




IDRIS-CNRS

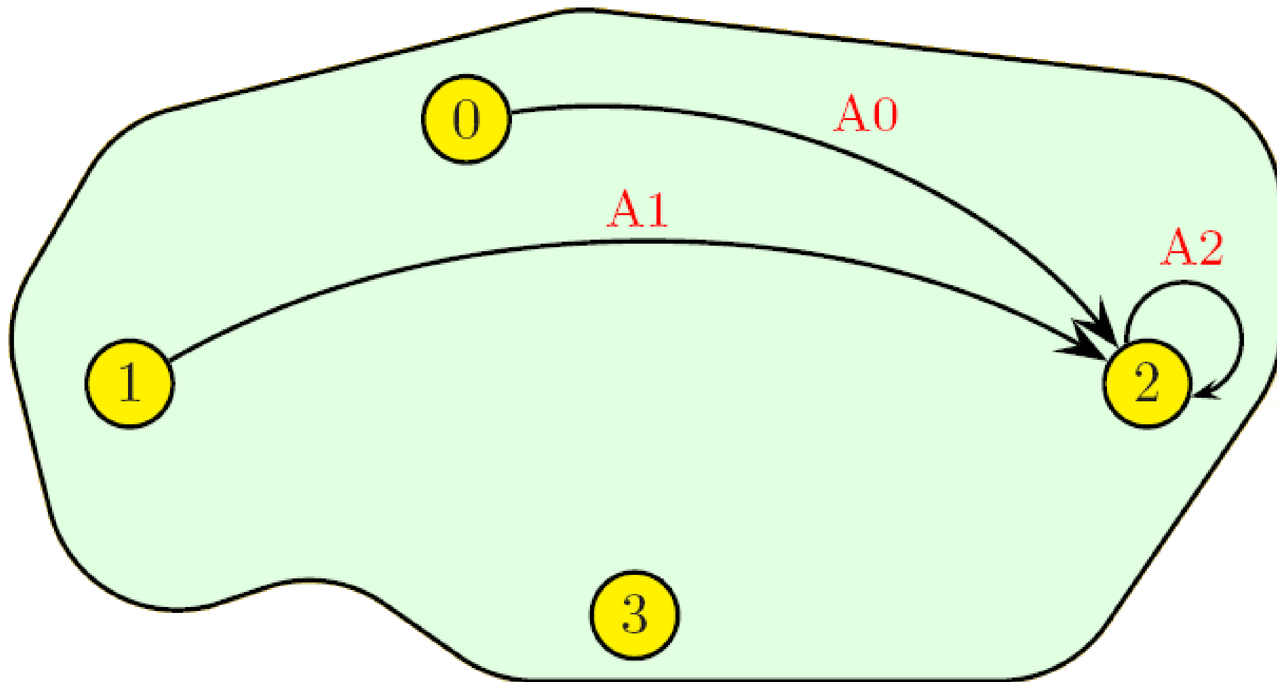
MPI_GATHER

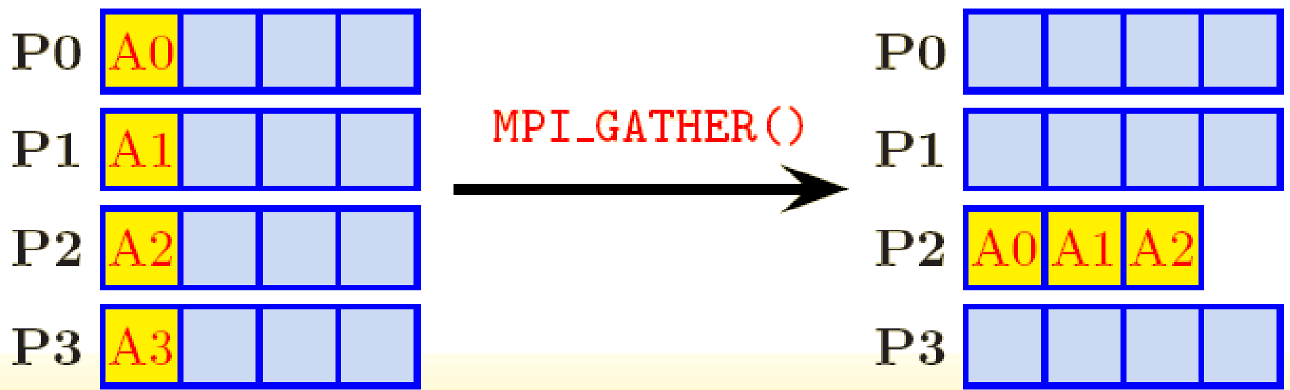




IDRIS-CNRS

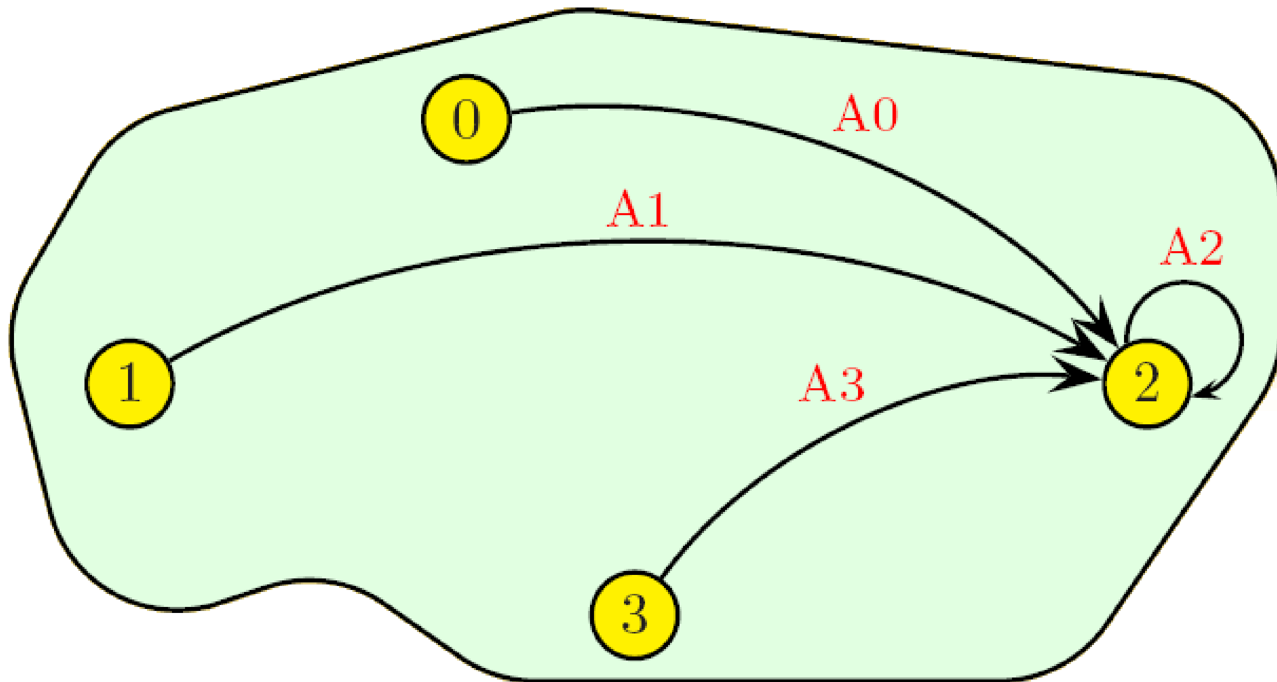
MPI_GATHER

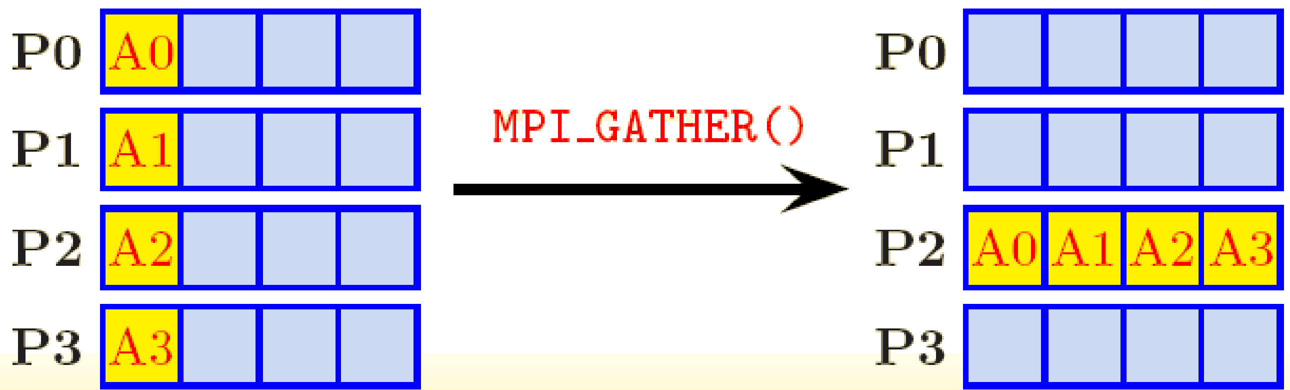




IDRIS-CNRS

MPI_GATHER



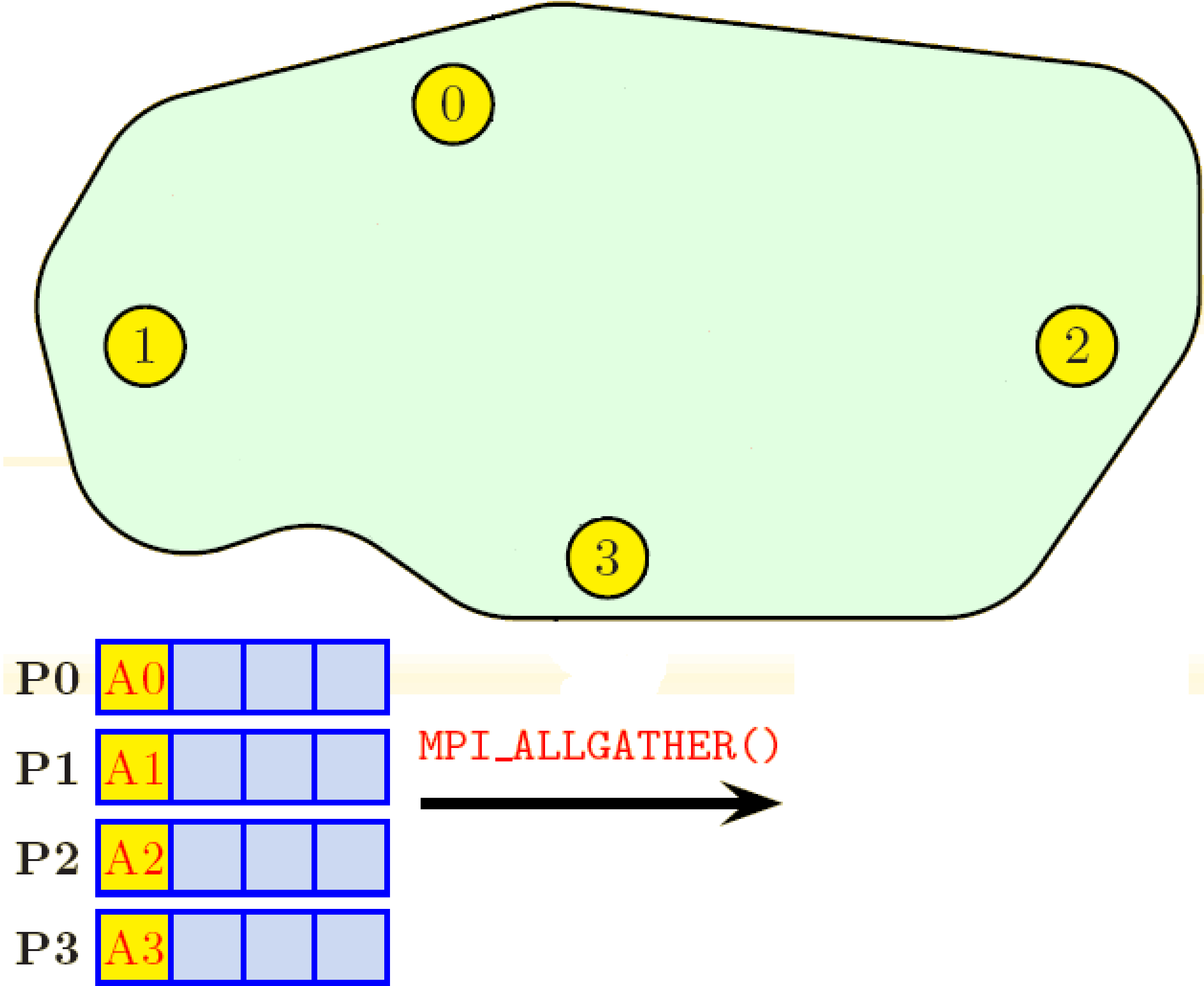


IDRIS-CNRS

Gather Routines

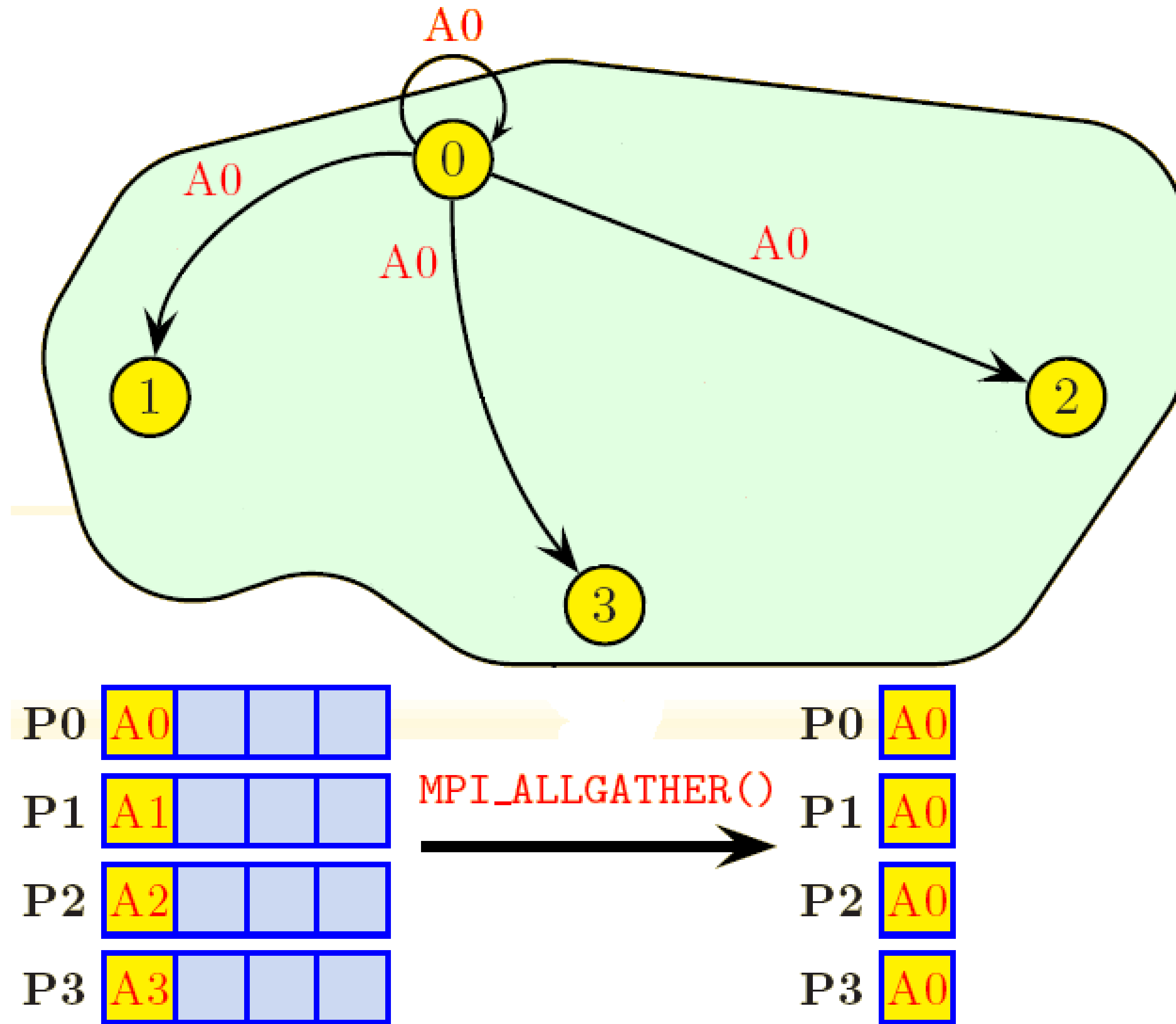
- **MPI_ALLGATHER**
  - gather arrays of equal length into one array on <u>all</u> tasks
  - Simpler and more efficient than doing MPI_GATHER followed by MPI_BCAST
- **MPI_GATHERV**
  - gather arrays of different lengths into one array on one task
- **MPI_ALLGATHERV**
  - gather arrays of different lengths into one array on <u>all</u> tasks
- **Where do you think these may be useful?**

MPI_ALLGATHER



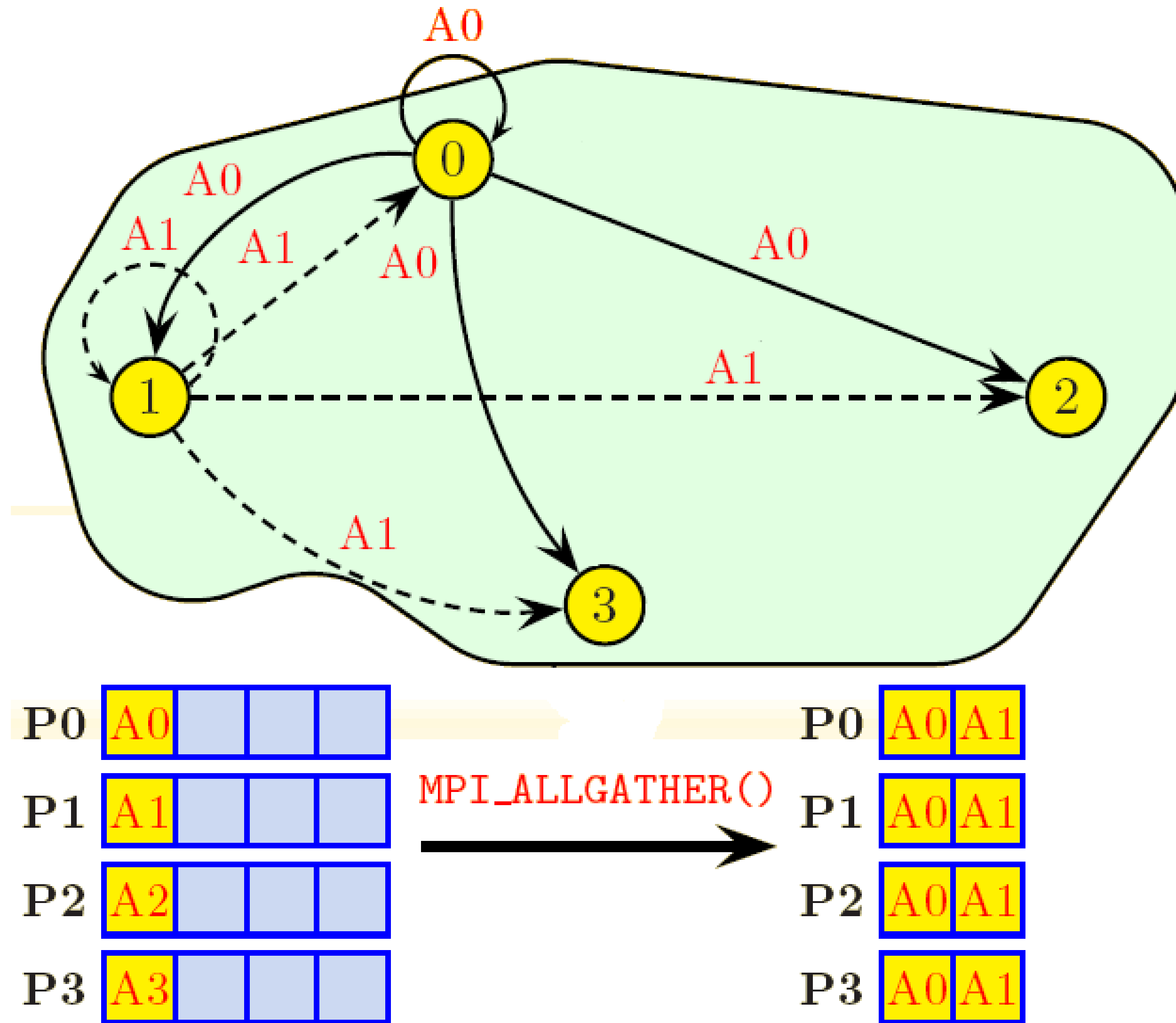
IDRIS-CNRS

MPI_ALLGATHER



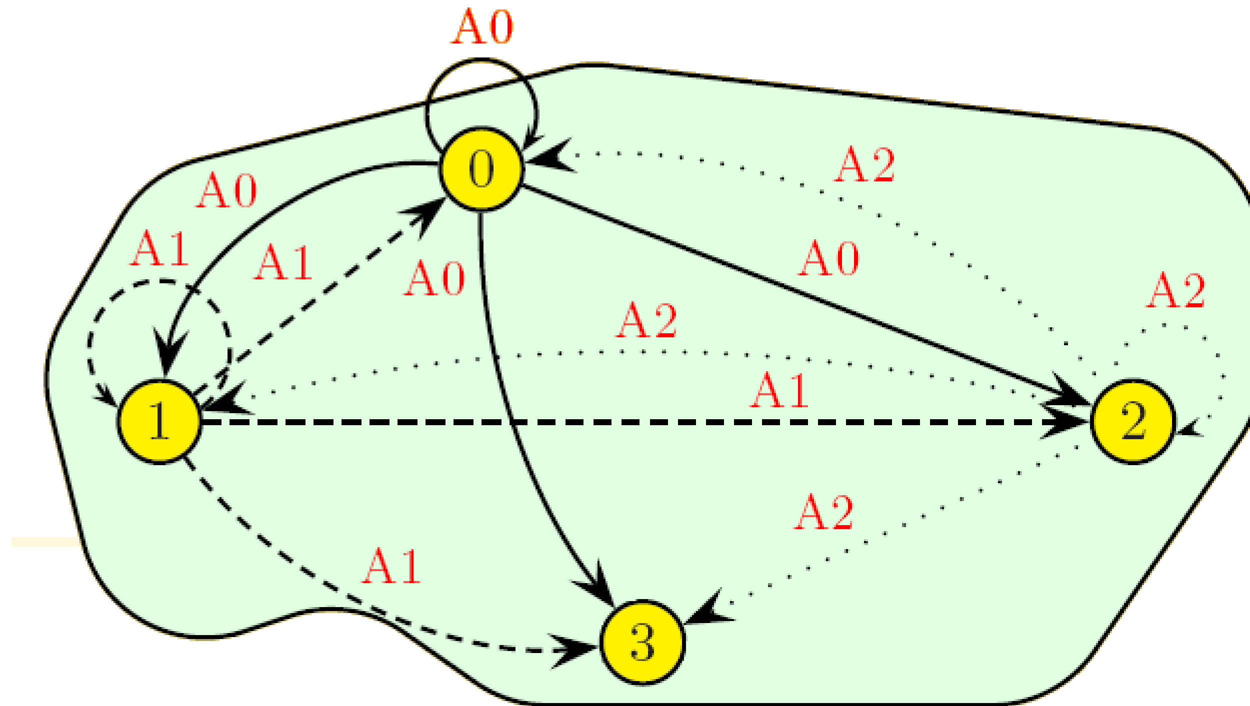
IDRIS-CNRS

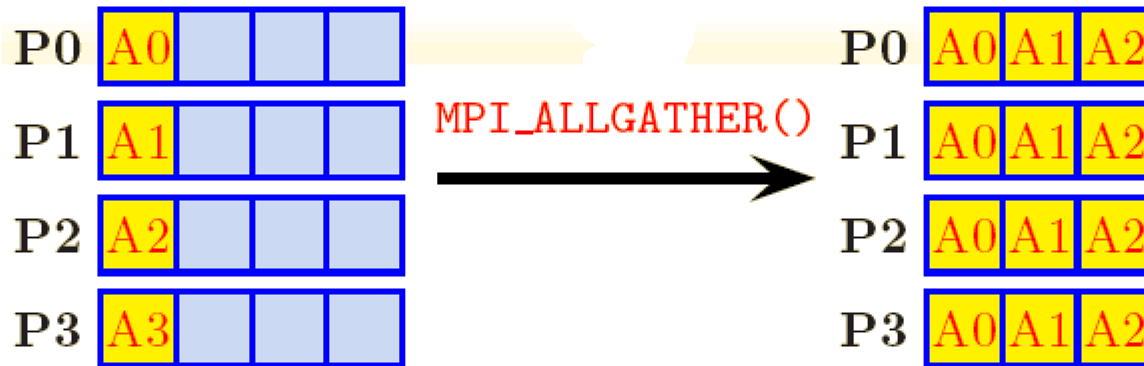
MPI_ALLGATHER



IDRIS-CNRS

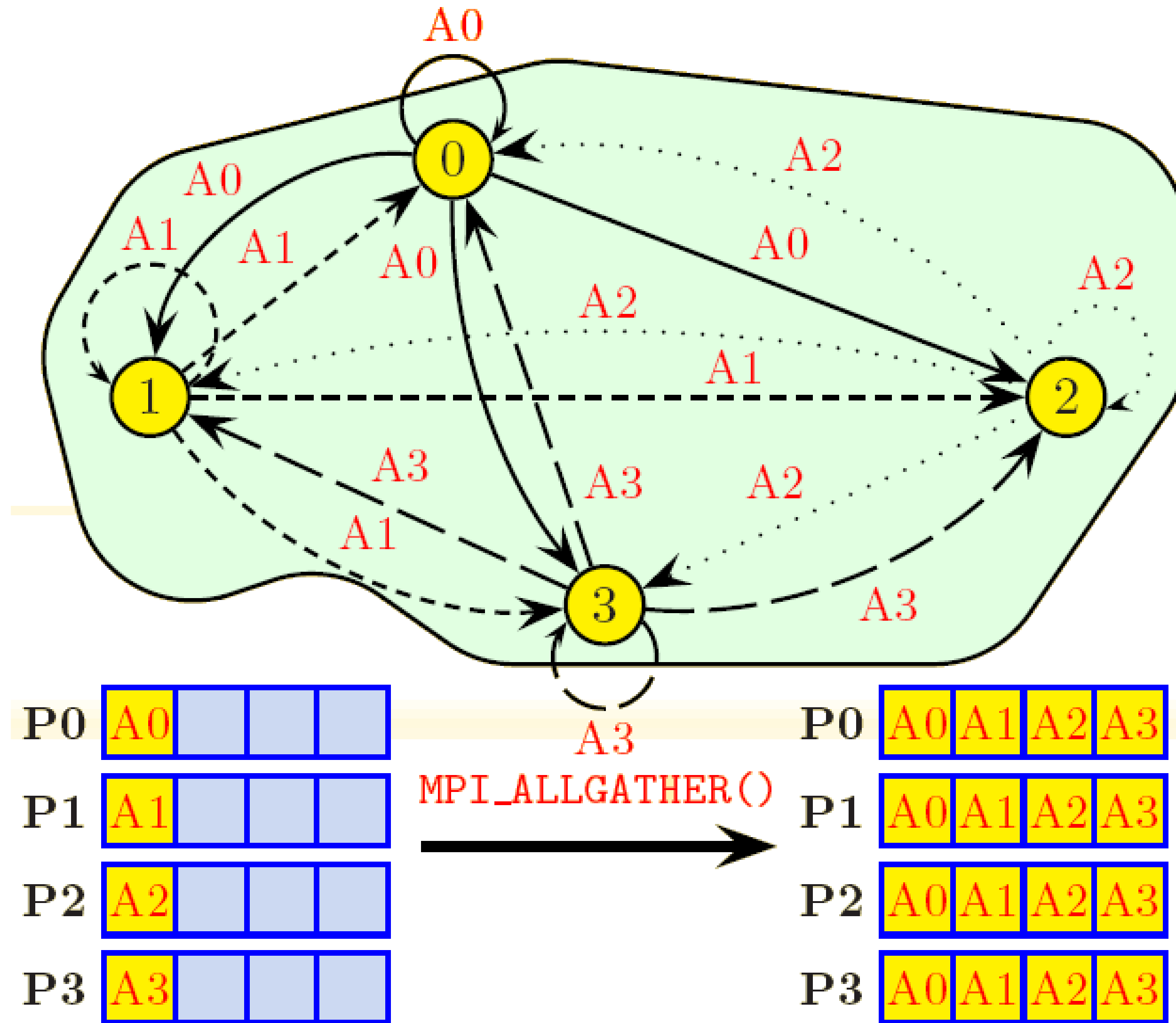
MPI_ALLGATHER





IDRIS-CNRS

MPI_ALLGATHER



IDRIS-CNRS

Scatter Routines

- `MPI_SCATTER`
  - divide one array on one task equally amongst all tasks
  - each task receives the same amount of data
- `MPI_SCATTERV`
  - divide one array on one task unequally amongst all tasks
  - each task can receive a different amount of data
- **Where do you think they might be useful?**

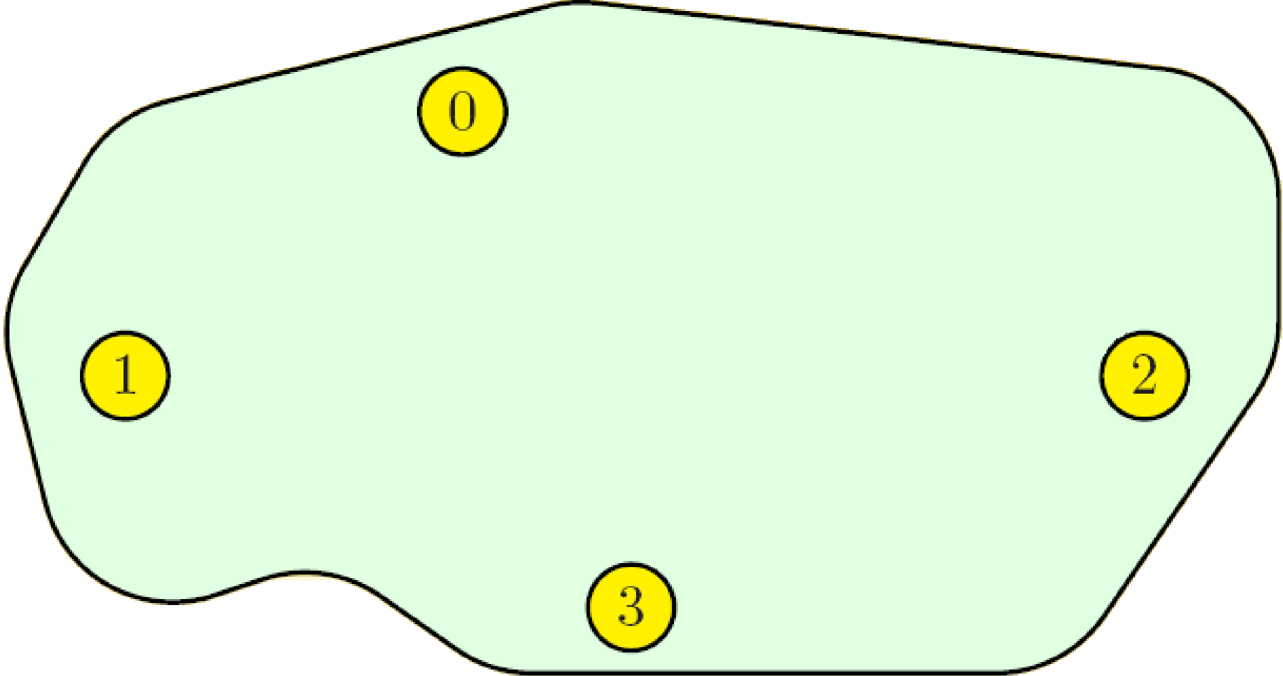
MPI_SCATTER

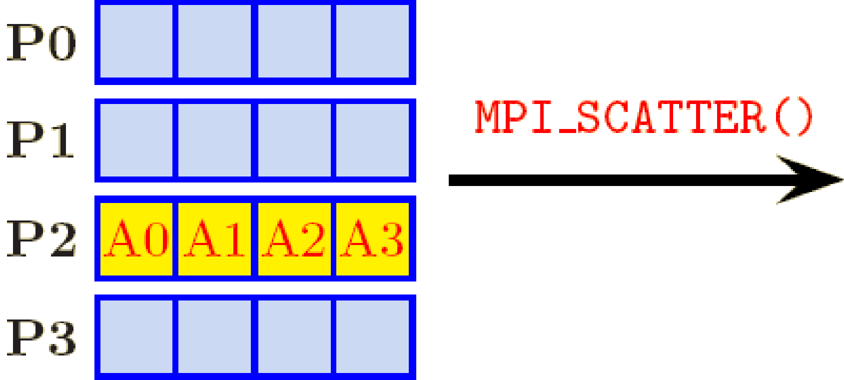
```
FORTRAN_TYPE:: sbuff, rbuff
integer:: count, root, ierror
call MPI_SCATTER( sbuff,scount,MPI_TYPE, &
                  rbuff,rcount,MPI_TYPE,root,MPI_COMM_WORLD,ierror)
```

- **ROOT** — task doing scatter — input
- **SBUFF** — array being sent — input
- **RBUFF** — array being received — output
- **[S/R]COUNT** — number of items to/from each task — input

The contents of `sbuff` on task id `root` are equally split and each task receives its part in array `rbuff`. Could also be done by putting `MPI_SEND` in a loop.

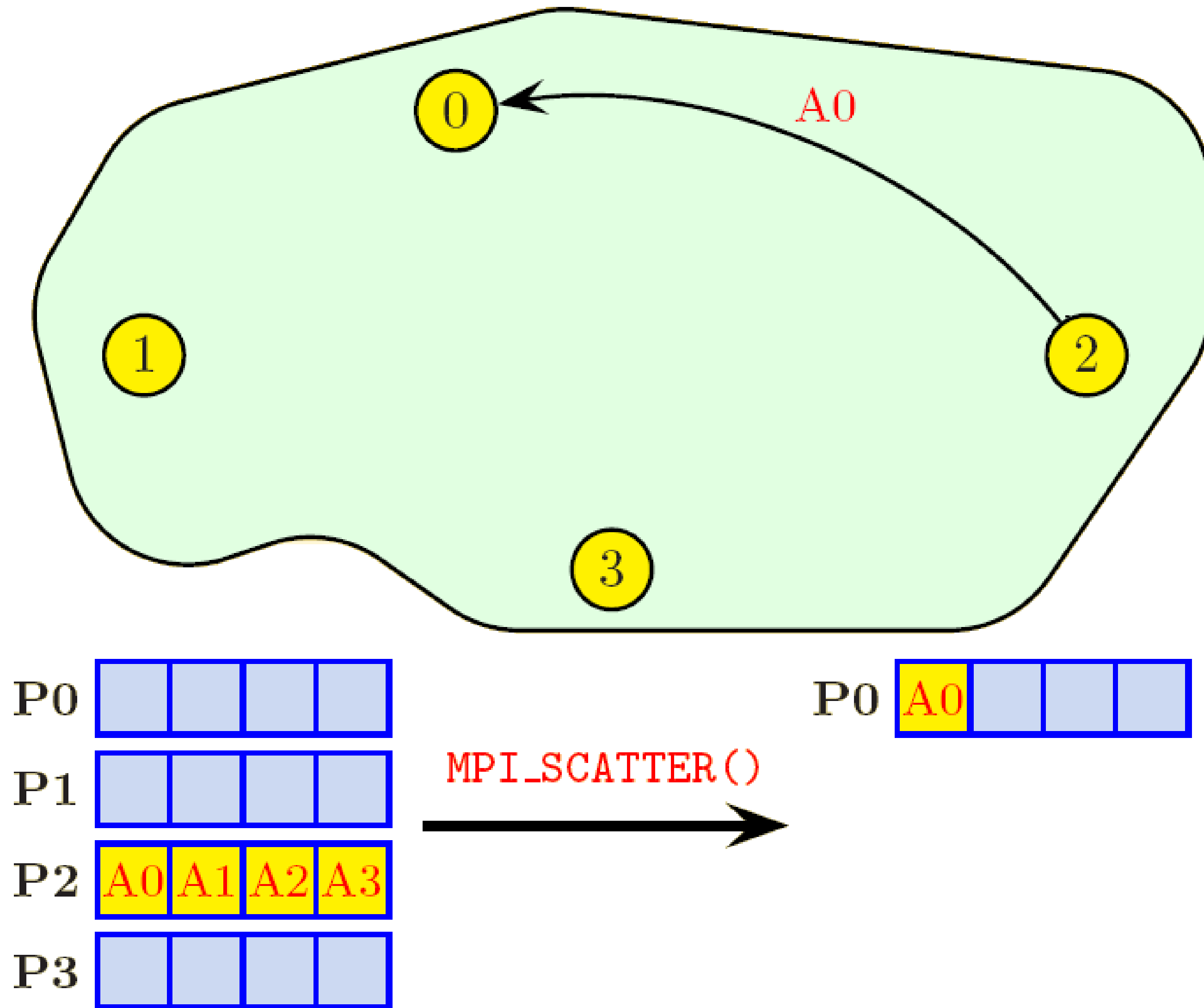
MPI_SCATTER





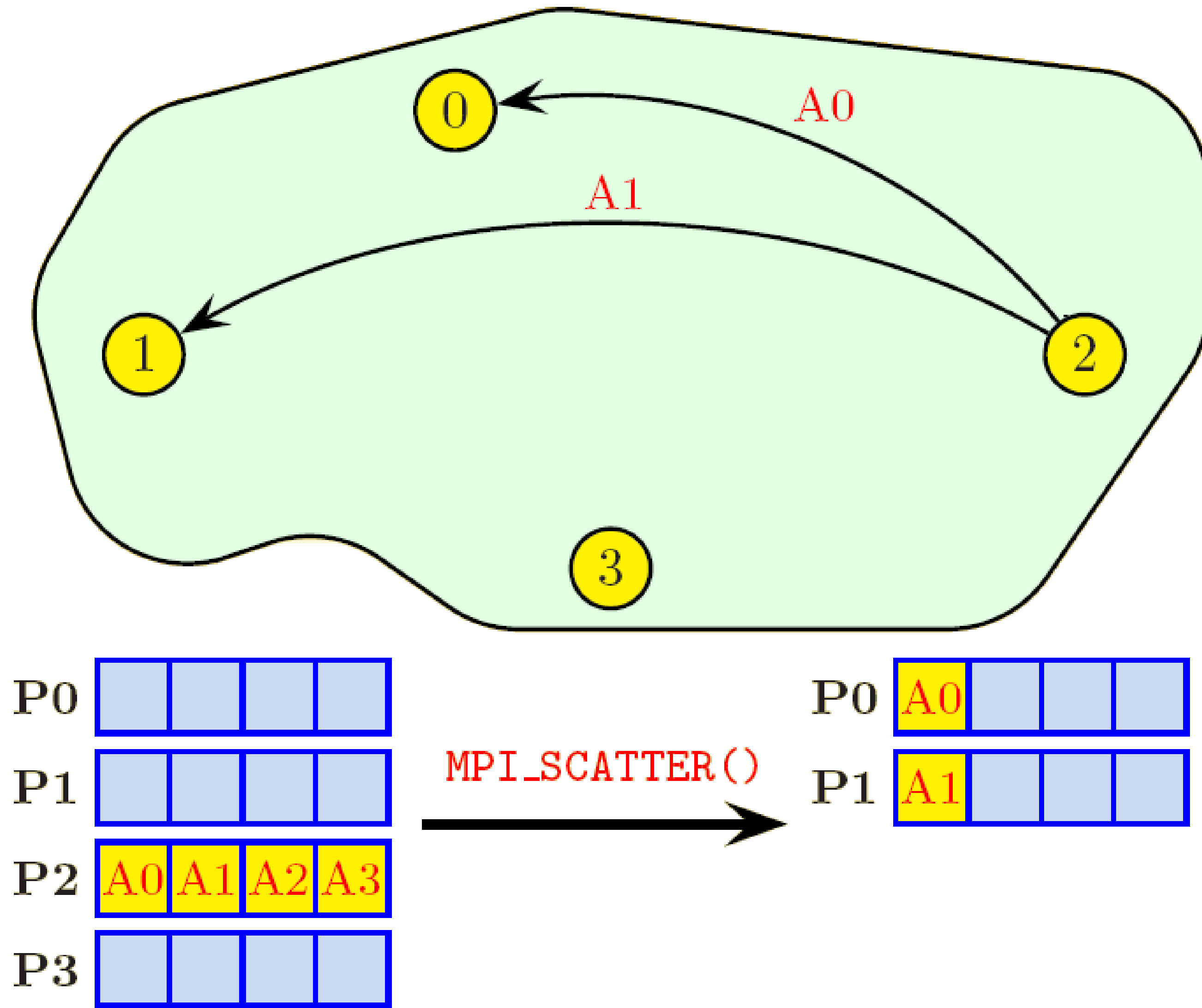
IDRIS-CNRS

MPI_SCATTER



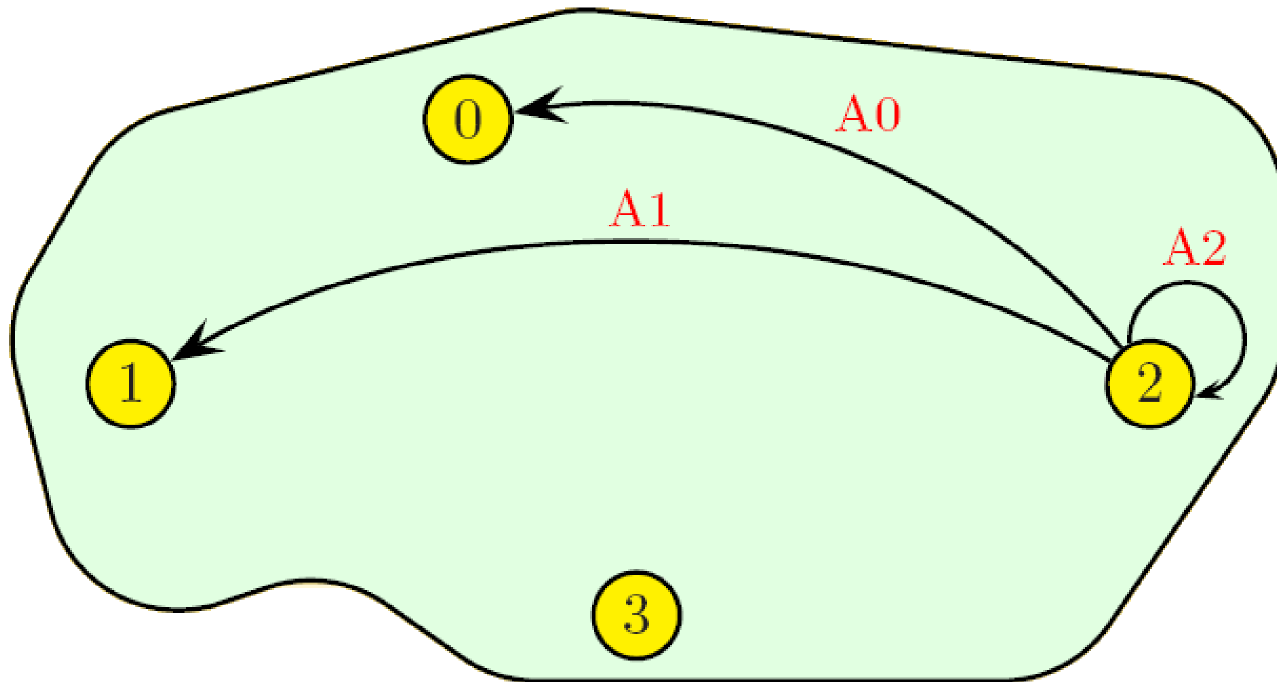
IDRIS-CNRS

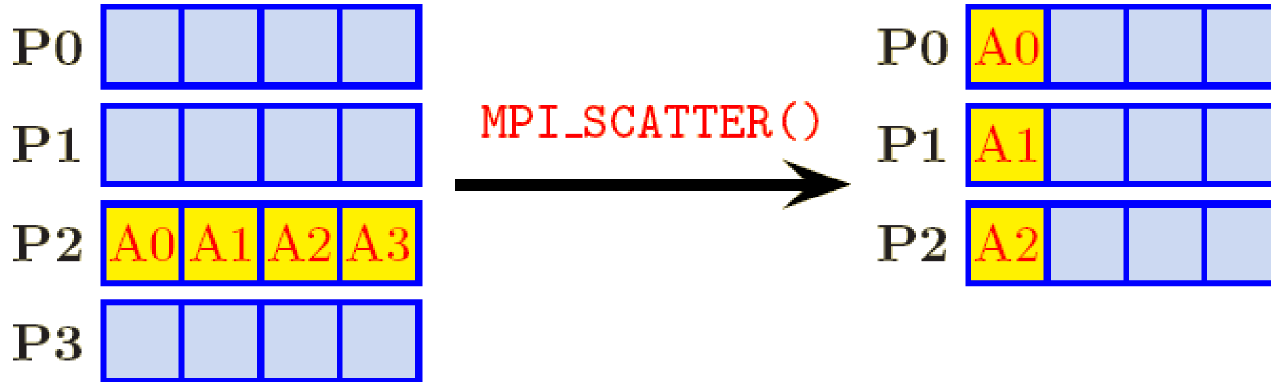
MPI_SCATTER



IDRIS-CNRS

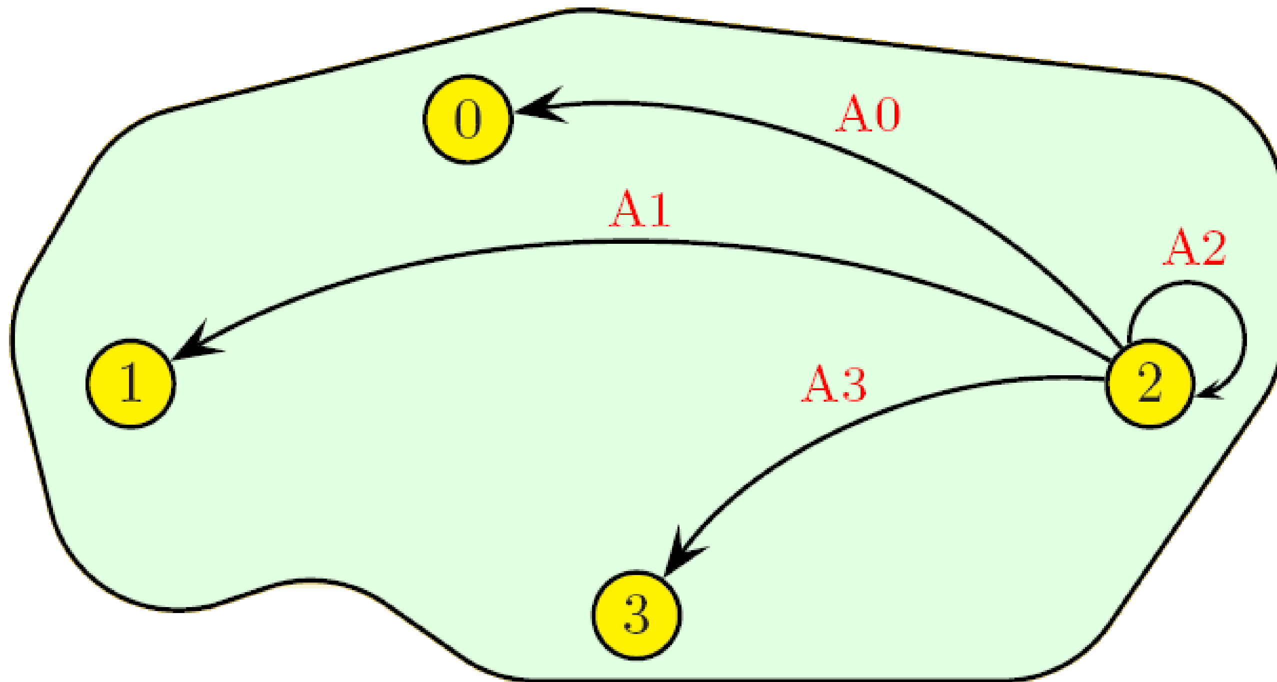
MPI_SCATTER

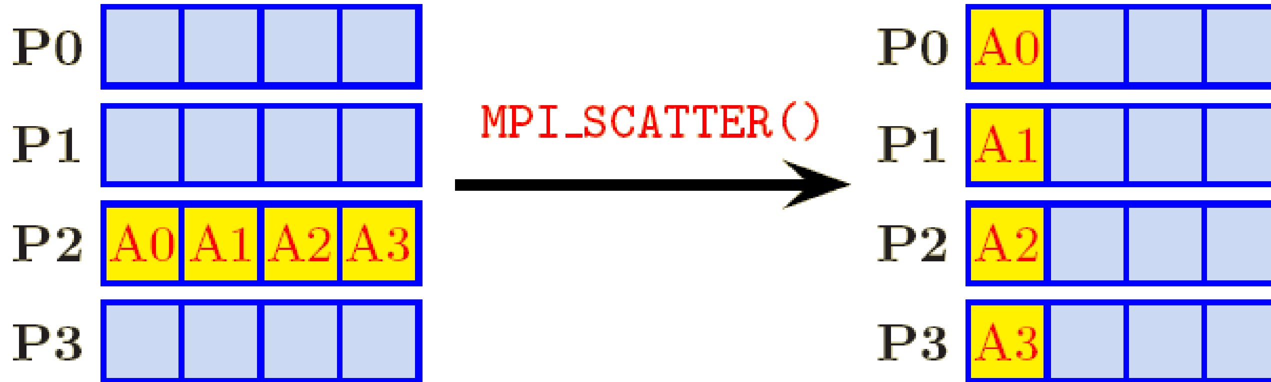




IDRIS-CNRS

MPI_SCATTER





IDRIS-CNRS

All to All Routines

- **MPI_ALLTOALL**
  - every task sends equal length parts of an array to all other tasks
  - every task receives equal parts from all other tasks
  - transpose of data over the tasks

- **MPI_ALLTOALLV**
  - as above but parts are different lengths

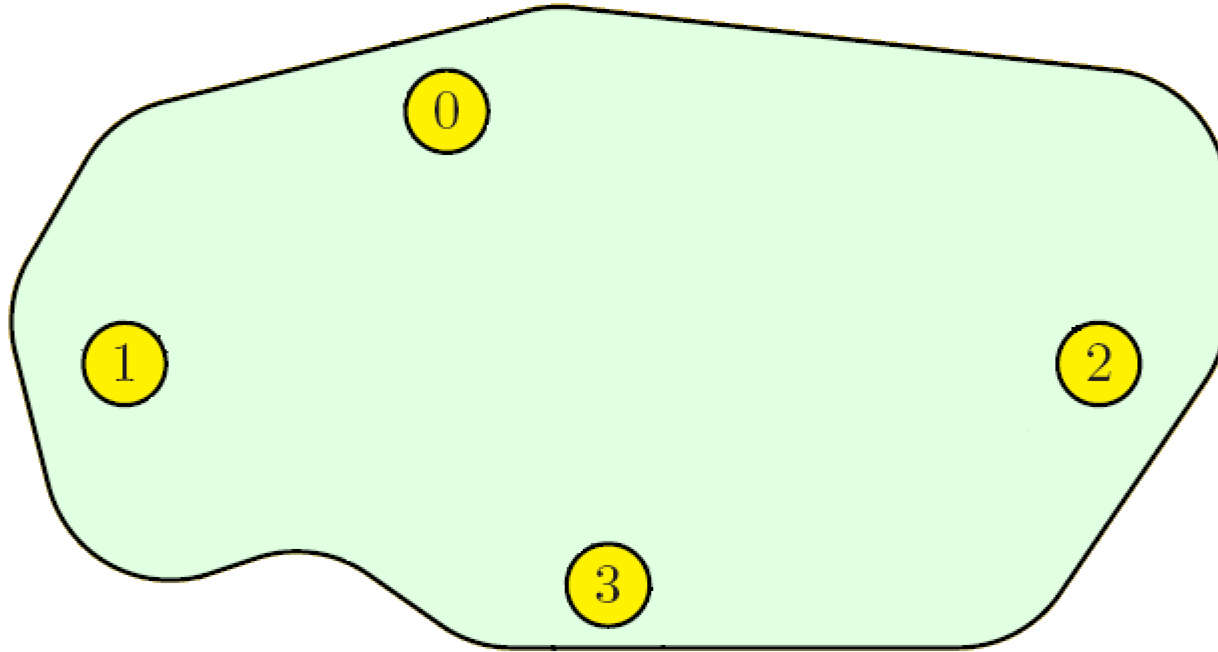
MPI_ALLTOALL

```
FORTRAN_TYPE:: sbuff, rbuff
integer:: count, root, ierror
call MPI_SCATTER( sbuff,scount,MPI_TYPE, &
                  rbuff,rcount,MPI_TYPE,MPI_COMM_WORLD,ierror)
```

- **SBUFF** — **array being sent** — **input**
- **RBUFF** — **array being received** — **output**
- **[S/R]COUNT** — **number of items to/from each task** — **input**

The contents of `sbuff` on each task are equally split and each task receives an equal part into array `rbuff`. Could also be done by putting `MPI_SEND/MPI_RECV` in a loop.

MPI_ALLTOALL

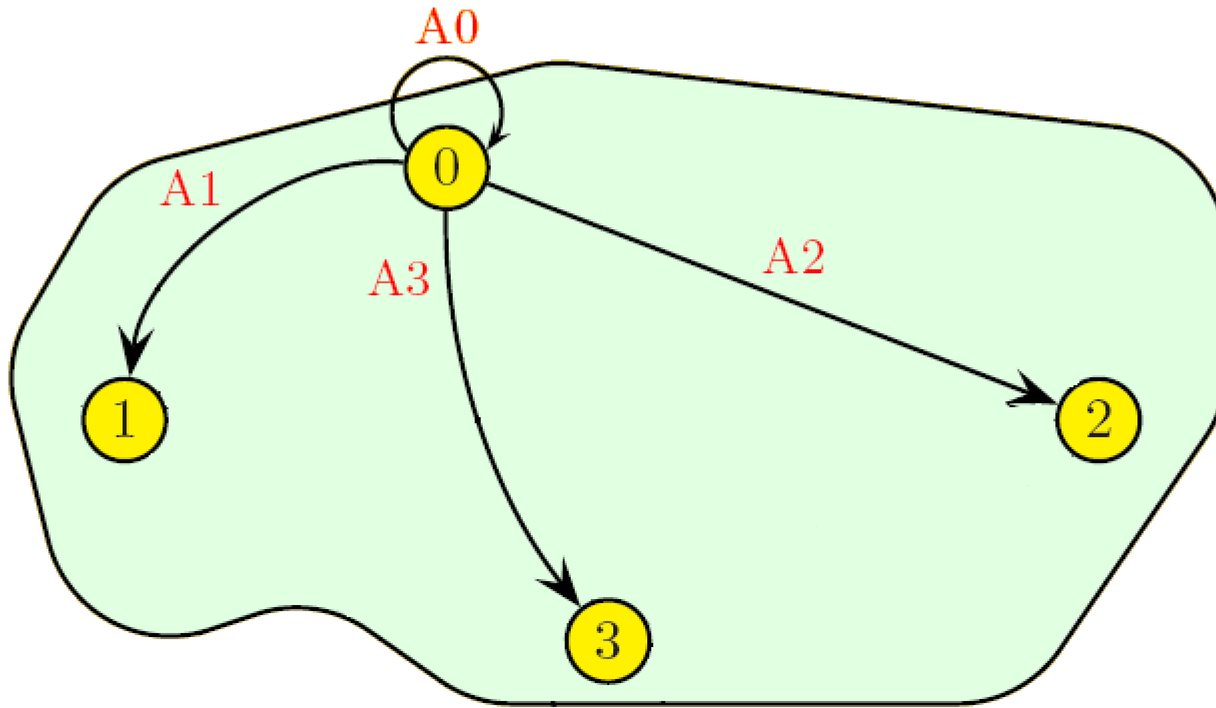


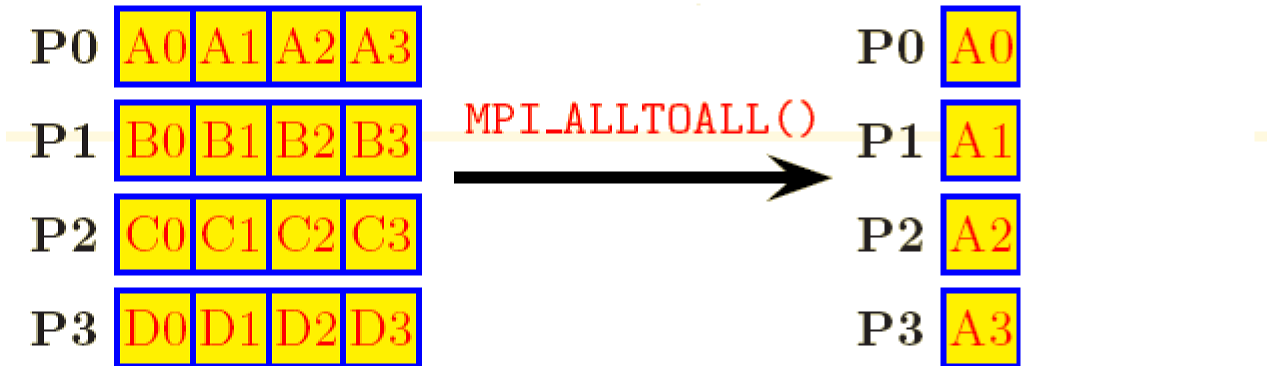
P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

MPI_ALLTOALL()

IDRIS-CNS

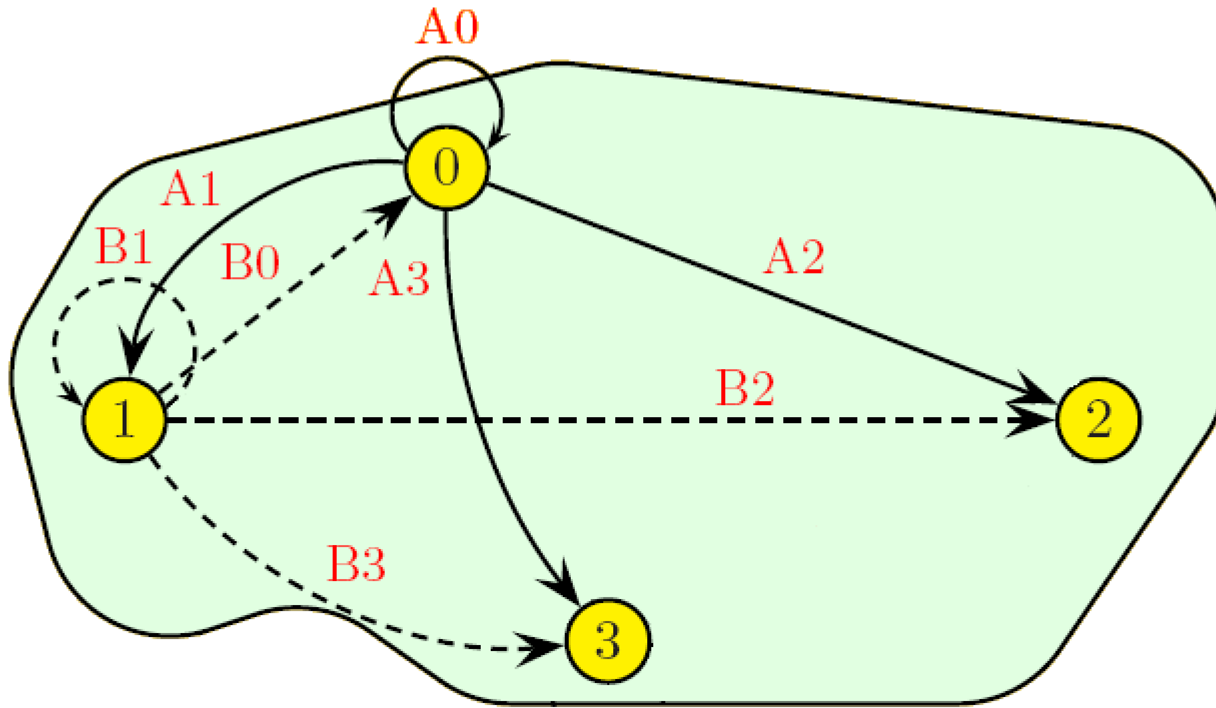
MPI_ALLTOALL

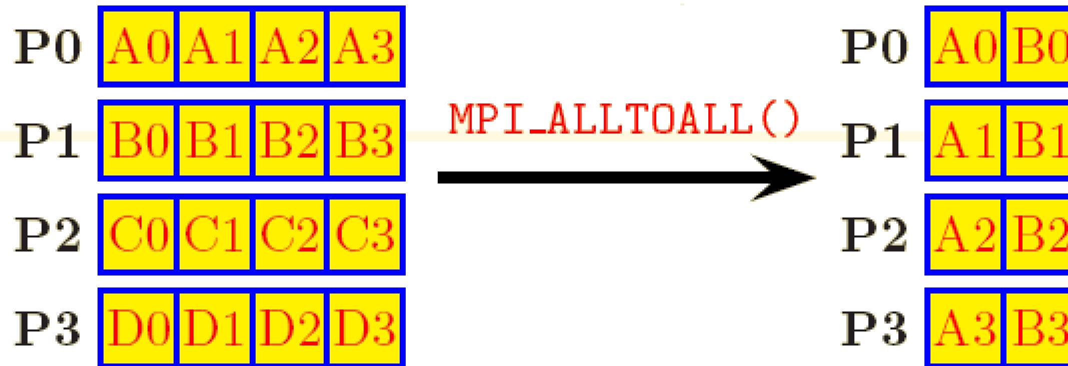




IDRIS-CNRS

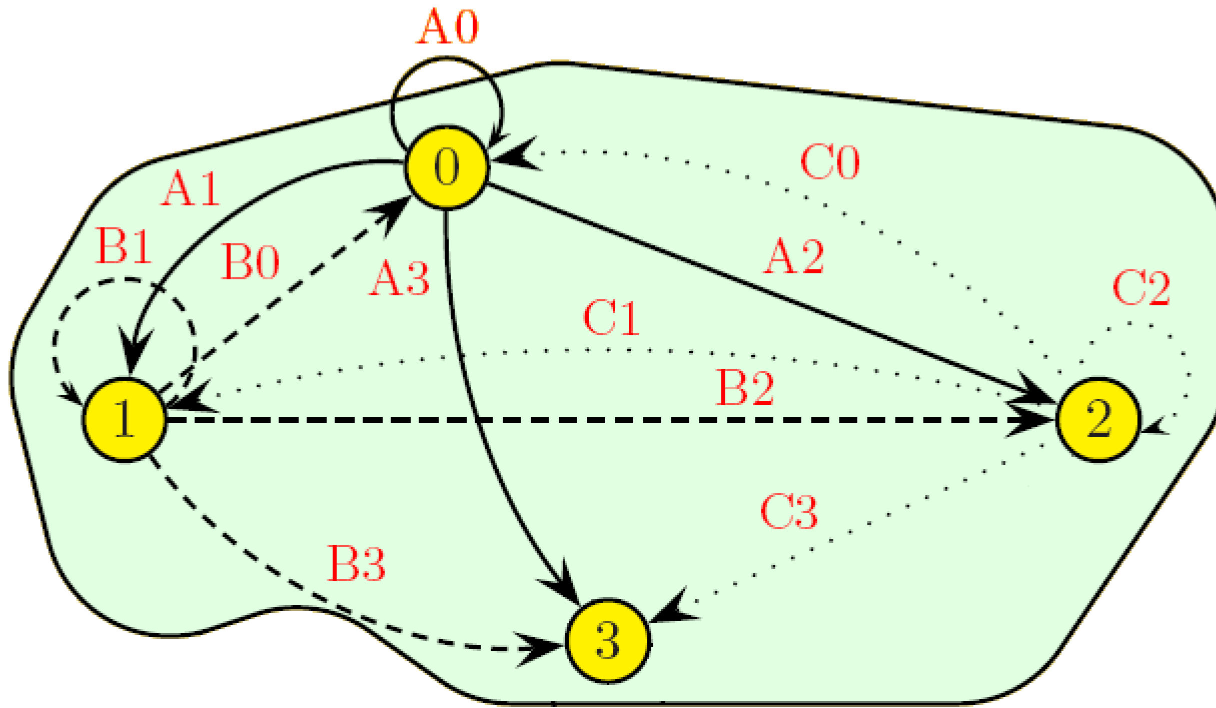
MPI_ALLTOALL

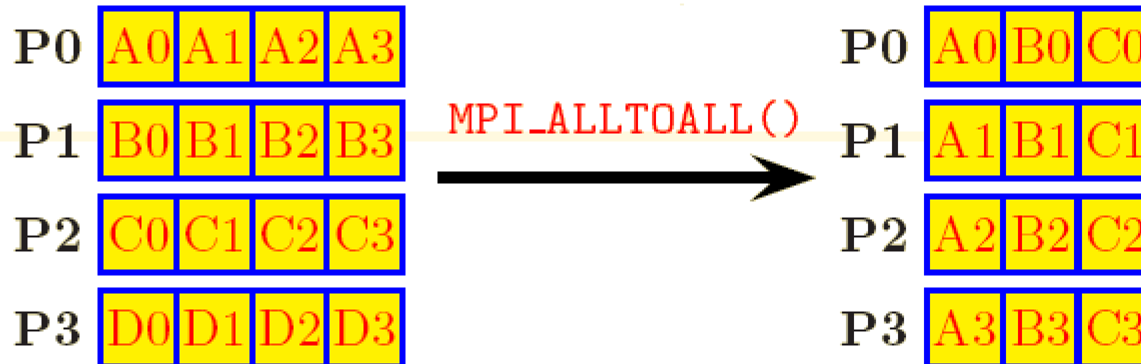




IDRIS-CNRS

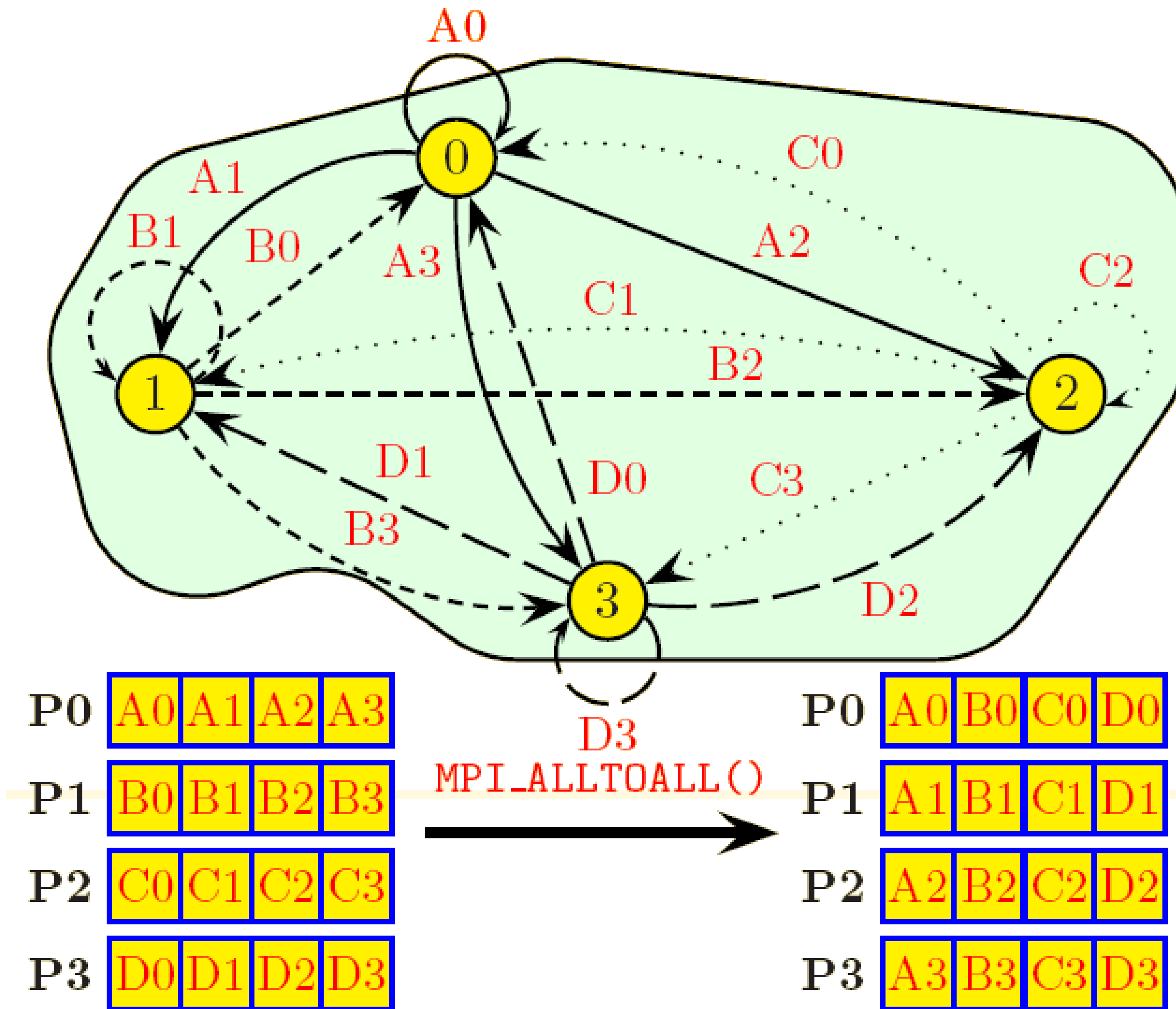
MPI_ALLTOALL





IDRIS-CNRS

MPI_ALLTOALL



IDRIS-CNRS

Reduction routines

- **Perform both communications and simple math**
  - Global sum, min, max, ...
- **Beware reproducibility**
  - MPI makes no guarantee of reproducibility
    - Eg. Summing an array of real numbers from each task
    - May be summed in a different order each time
  - You may need to write your own order preserving summation if reproducibility is important to you.
- **`MPI_REDUCE`**
  - every task sends data and result is computed on the "root" task
- **`MPI_ALLREDUCE`**
  - every task sends, result is computed and broadcast back to all tasks. Equivalent to `MPI_REDUCE` followed by `MPI_BCAST`

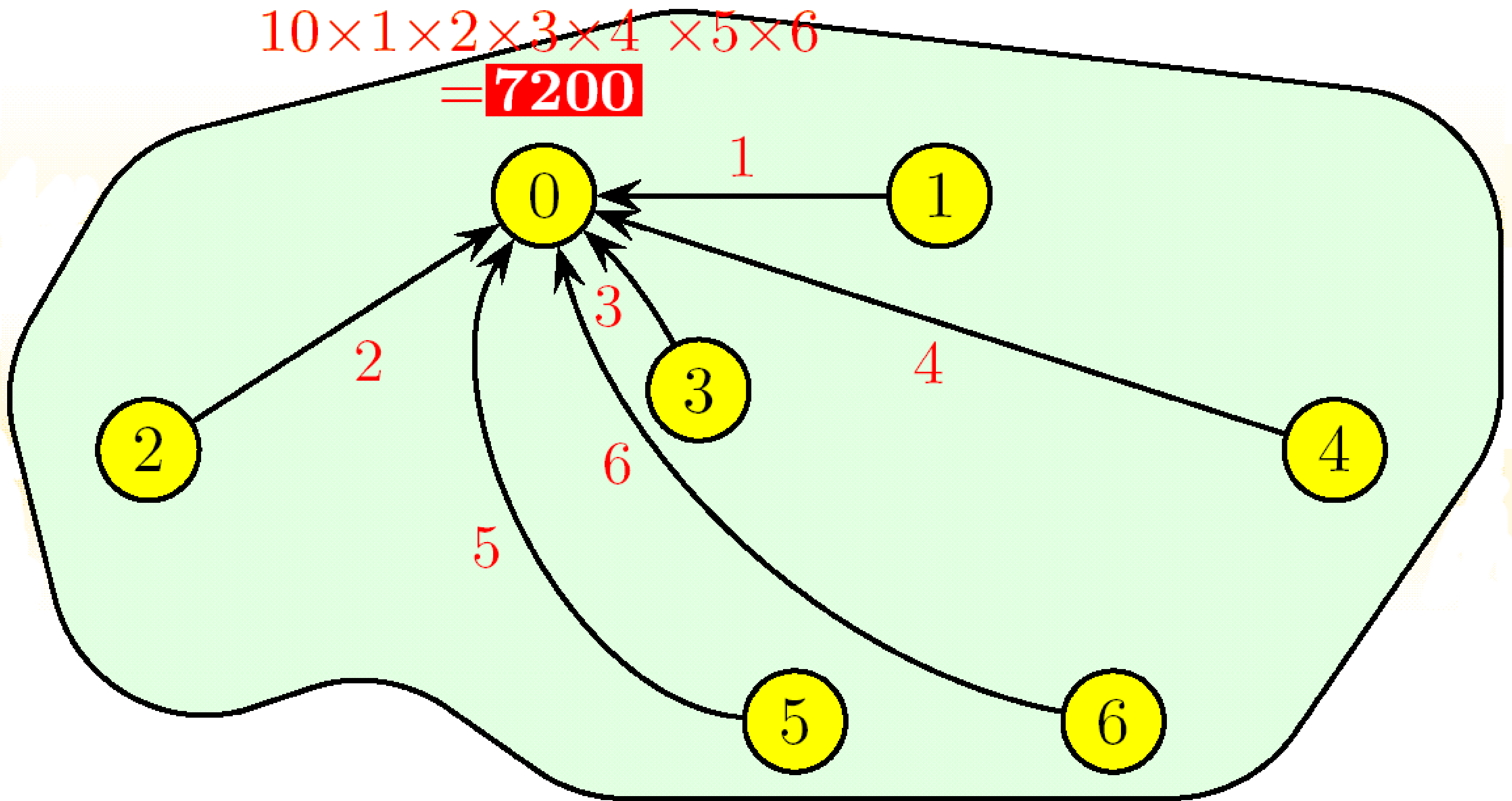
MPI_REDUCE

```
FORTRAN_TYPE:: sbuff, rbuff
integer:: count, root, ierror
call MPI_REDUCE( sbuff,rbuff,count,MPI_TYPE,OP_TYPE, &
                 root,MPI_COMM_WORLD,ierror)
```

• **SBUFF**	**array to be reduced**	**input**
• **RBUFF**	**result of reduction**	**output**
• **COUNT**	**number of items to be reduced**	**input**

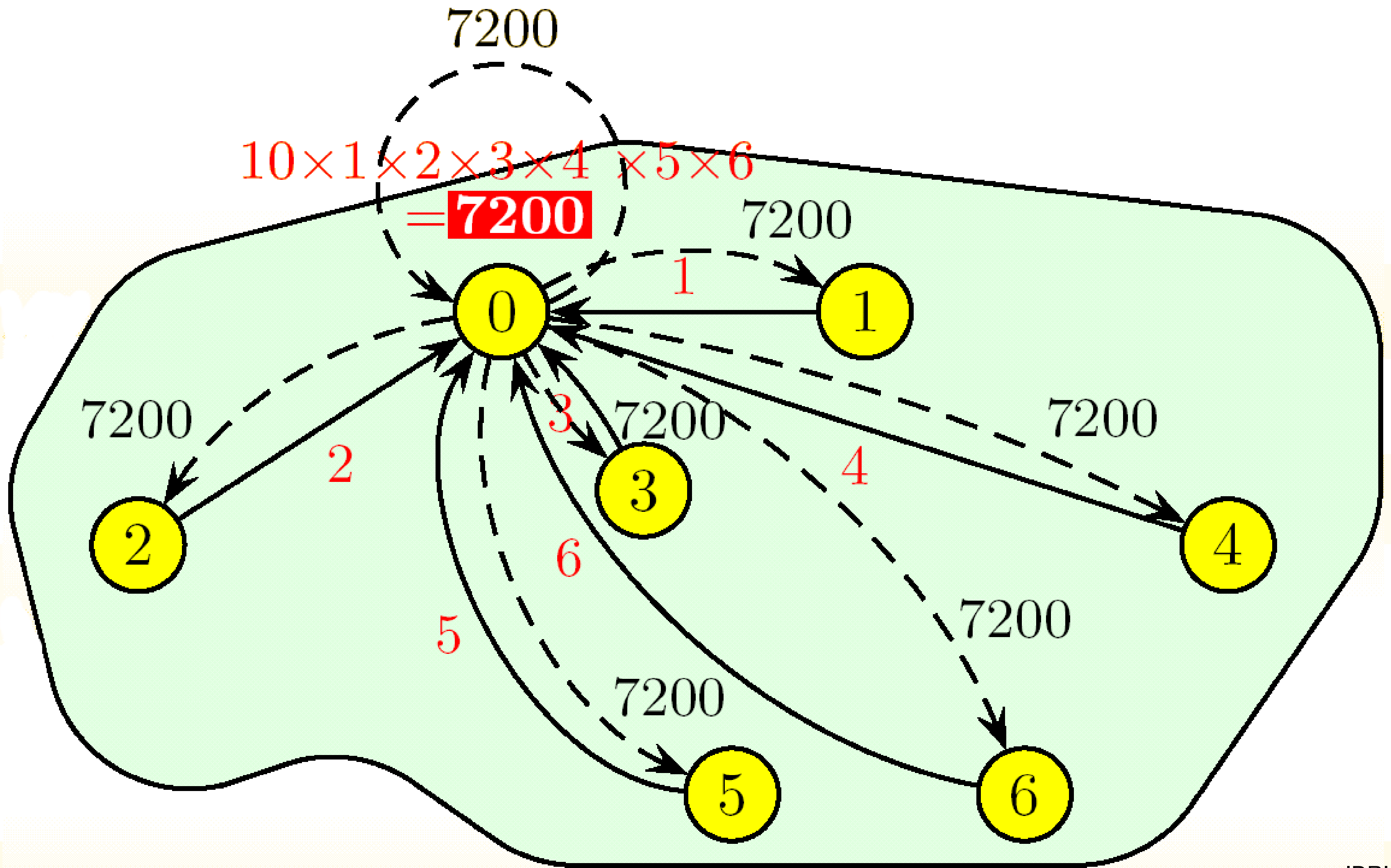
The contents of `sbuff` from all tasks are reduced according to `OP_TYPE` and the result is sent to RBUFF task `root`. `OP_TYPE` can be `MPI_MAX, MPI_MIN, MPI_SUM, MPI_IPROD, MPI_IAND, MPI_BAND, MPI_IOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC`

MPI_REDUCE



IDRIS-CNRS

MPI_ALLREDUCE



MPI References

- **Using MPI (2nd edition) by William Gropp, Ewing Lusk and Anthony Skjellum; Copyright 1999 MIT; MIT Press ISBN 0-262-57132-3**
- **The Message Passing Interface Standard on the web at**

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