An Introduction to MPI Programming

Paul Burton

March 2010

Topics

- Introduction
- Initialising MPI
- Data Types and Tags
- Basic Send/Receive

- Practical 1

- Receive Part 2
- Collective CommunicationsReferences

- Practical 2

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 (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
- **Works within SMP and across Distributed Memory Nodes**
- **Can scale to hundreds 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 on the ECMWF IBM**
- **A mass of useful material on the Web**

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**

Some definitions

- **Task**
  - one running instance (copy) of a program
  - same as a process
  - IBM Loadleveler talks about tasks not processes
  - Basic unit of an MPI parallel execution
- **Master**
  - the master task is the first task in a parallel program
  - task id is 0
- **Slave**
  - all other tasks in a parallel program

The simplest MPI program.....

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

Hello World with MPI

```
program hello

use mpi
implicit none

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
```

MPIF.H

```
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 IBM's mpi to see if it does!**
- **In Fortran77 use "include 'mpif.h'" instead**

Hello World with MPI

```
program hello

use mpi
implicit none

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 IBM all tasks execute the code before MPI_INIT**
  - **this is an implementation dependent feature**

Hello World with MPI

```
program hello

use mpi
implicit none

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 subsets of tasks**
  - **dividing programs into subsets of tasks often not necessary**
  - **IFS also creates and uses some additional communicators**
    - **useful when doing collective communications**
  - **advanced topic**
- **MPI_COMM_WORLD means all tasks**
  - **most MPI programs just use MPI_COMM_WORLD**

Hello World with MPI

```
program hello

use mpi
implicit none

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 ntasks**
  - **the number of tasks is defined in a loadleveler directive**
- **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

use mpi
implicit none
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

use mpi
implicit none

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**
- **Does not stop the task**

MPI_ABORT

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

- **Causes all tasks to abort**
- **Even if only one task makes call**
- **Input : error_code**
  - **User defined error code**
- **Output : ierror**
  - **MPI returned status – did MPI_ABORT fail?!**

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**
  - data can be of various Fortran types
  - data length can be zero bytes to MB's
  - messages have tag identifiers
- **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)**
  - **permits send/receive to non contiguous buffers**
  - **advanced topic**

MPI Tags

- **All messages are given an integer TAG value**
  - **standard says maximum value is at least 32768 (2^15)**
- **This helps to identify a message**
- **Used to ensure messages are read in the right order**
  - **standard says nothing about the order of message arrival**
- **You decide what tag values to use**
  - **Enables you to keep track of multiple messages**
  - **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	**the kind of variable 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	**the kind of variable eg MPI_REAL**	**input**
SOURCE	**the task id of the sender**	**input**
TAG	**the message identifier**	**input**
STATUS	**information about the message**	**output**

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
```

Compiling an MPI Program

- **Use mpxlf90_r compiler**
  - **automatically finds mpi “use” file and loads appropriate libraries**

```
$ mpxlf90_r -c hello.f
$ mpxlf90_r hello.o -o hello
```

Loadleveler and MPI

- **Define your task requirements as loadleveler directives**

```
#@ job_type = parallel
#@ class = np
#@ network.MPI = csss,,us

#@ node = 2
#@ total_tasks = 64

or

#@ node = 2
#@ tasks_per_node = 32
```

First Practical

- **Copy all the practical exercises to your account:**
  - **cd $HOME**
  - **mkdir mpi_course ; cd mpi_course**
  - **cp –r ~trx/mpi.2010/* .**
- **Exercise1a**
  - **A simple message passing exchange based on “hello world”**
- **See the README for details**

More on MPI_RECV

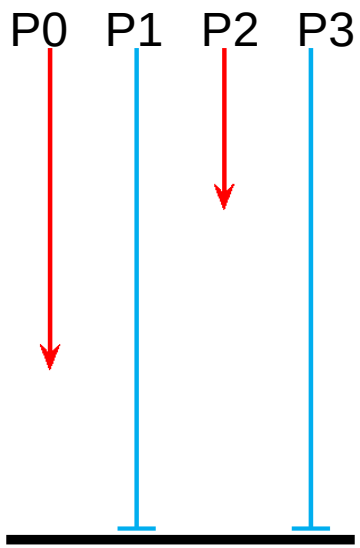
- **MPI_RECV will block waiting for the message**
  - **if message never sent then deadlock**
    - **task will wait until it hits cpu limit**
- **The source and tag can be less specific**
  - **MPI_ANY_SOURCE means any sender**
  - **MPI_ANY_TAG means any tag**
  - **Used to receive messages in a more random order**
  - **helps smooth out load imbalance**
  - **May require over-allocation of receive buffer**
- **status(MPI_SOURCE) will contain the actual sender**
- **status(MPI_TAG) will contain the actual tag**

MPI_BARRIER

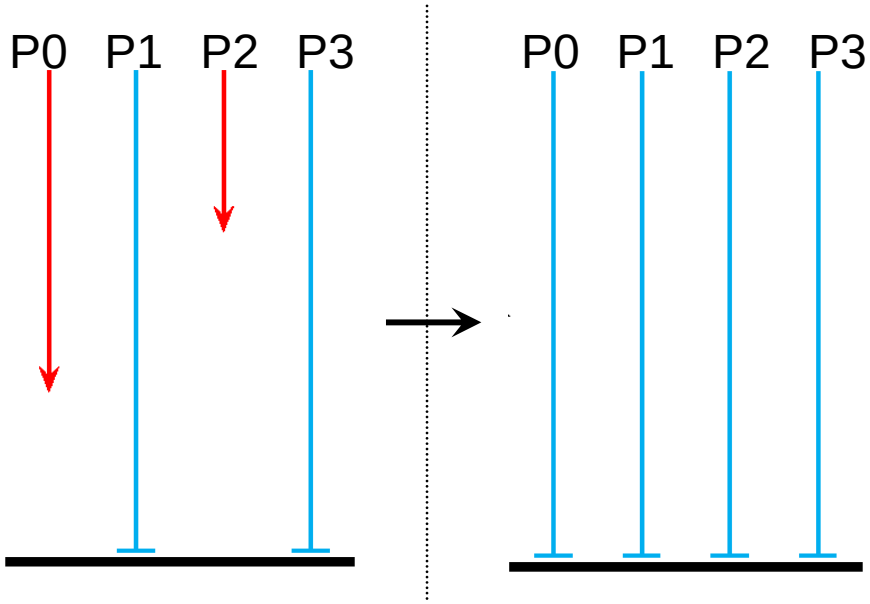
```
integer:: ierror
call MPI_BARRIER(MPI_COMM_WORLD,ierror)
```

- **Forces all tasks to synchronise**
  - **for timing points**
  - **to improve output of prints**
  - **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**
  - **program will loop until it hits cpu limit**

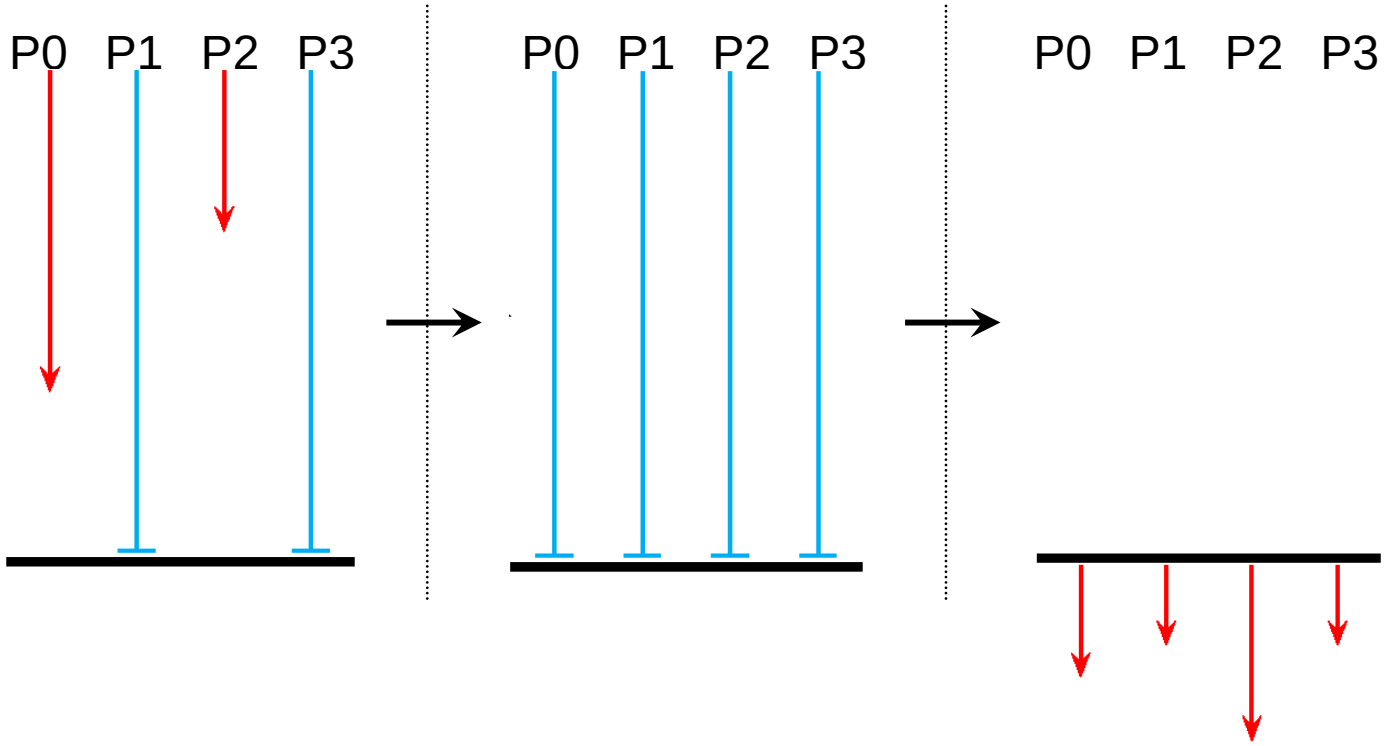
MPI_BARRIER



MPI_BARRIER



MPI_BARRIER



Collective Communications

- **MPI contains Collective Communications routines**
  - **called by all tasks 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_Broadcast and MPI_Gather**
- **Other routines will be summarised**
- **IFS uses some collective routines**

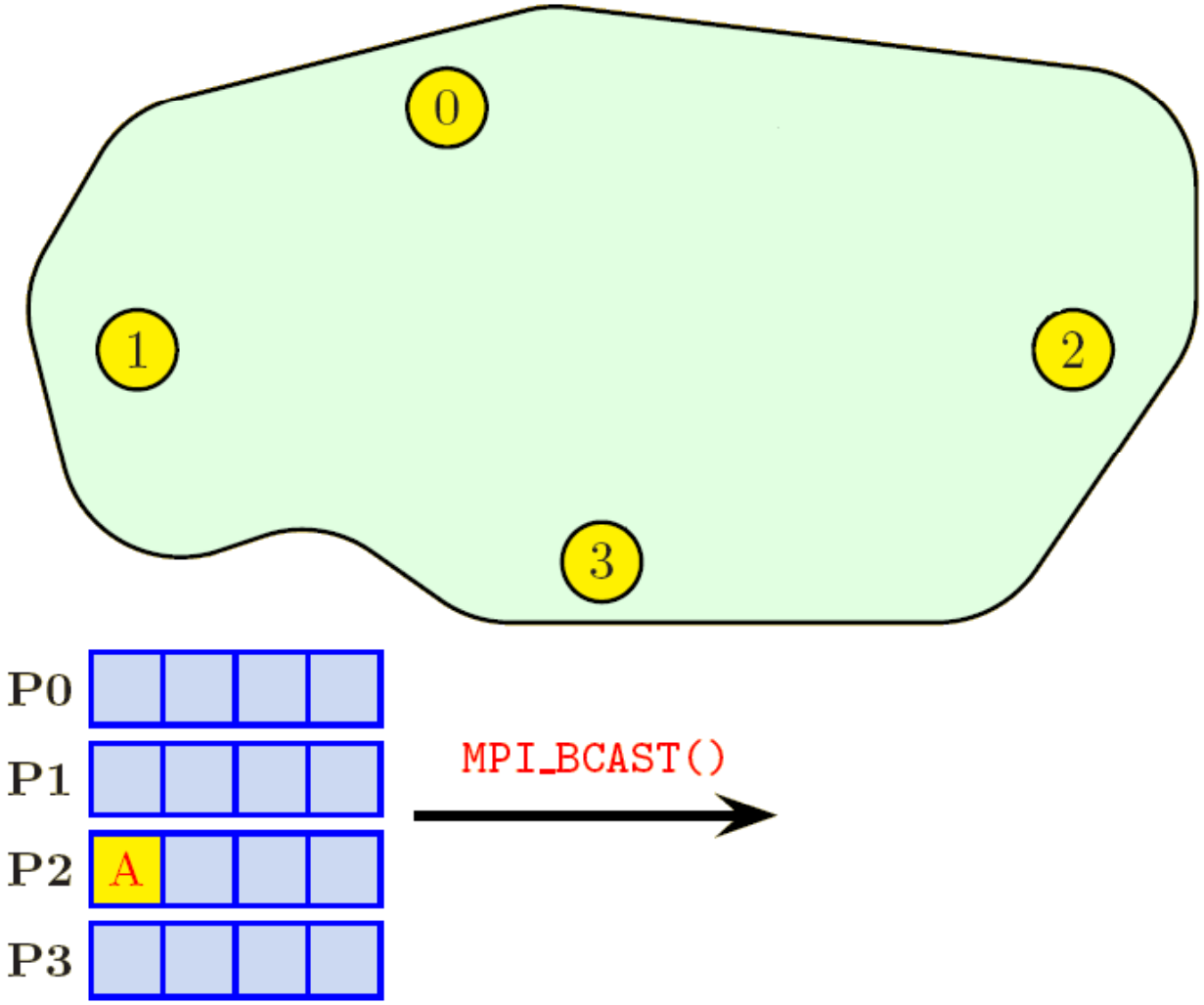
MPI_BROADCAST

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

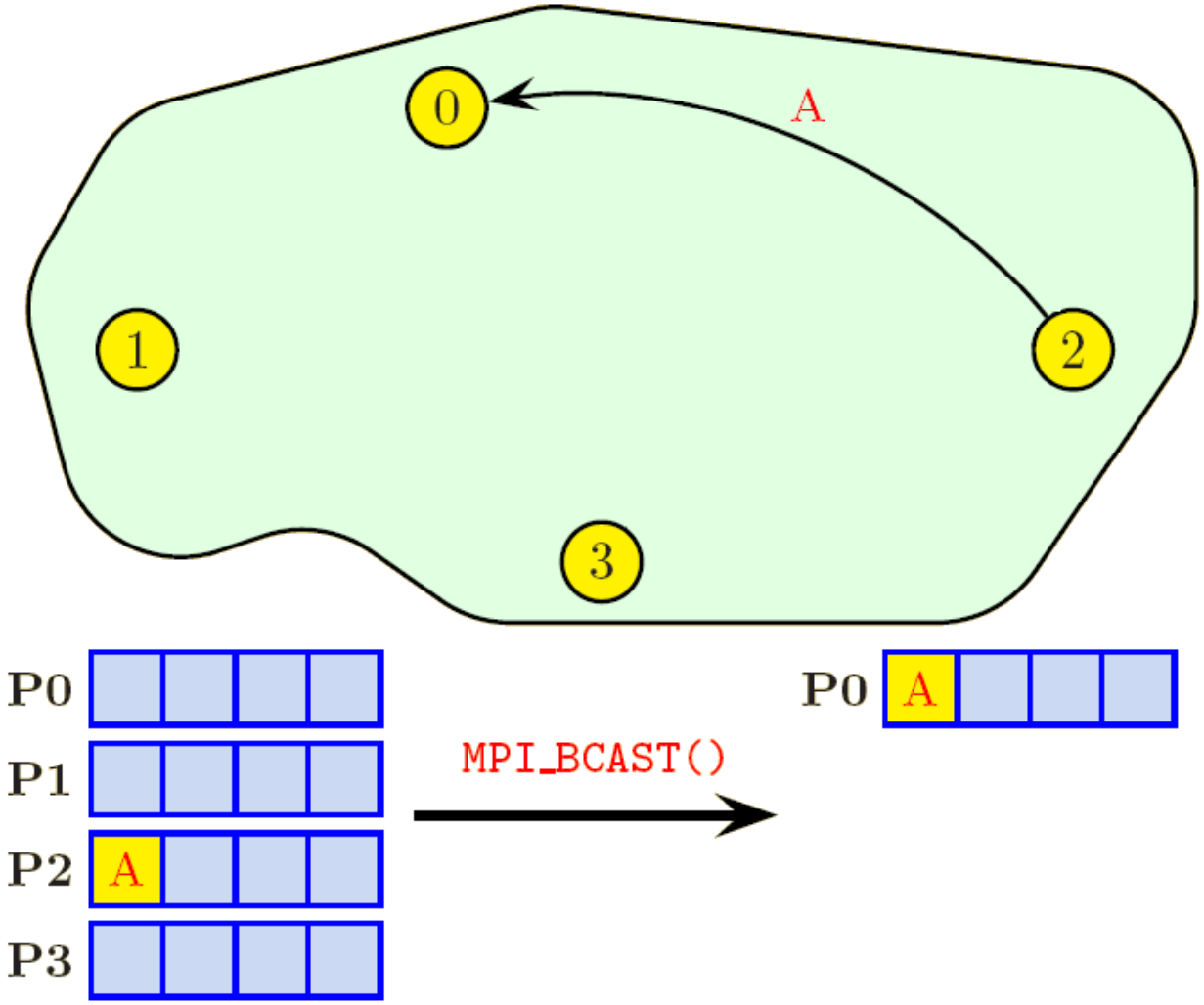
ROOT	**task doing broadcast**	**input**
BUFF	**array being broadcast**	**input/output**
COUNT	**the number of elements**	**input**
MPI_TYPE	**the kind of variable**	**input**

The contents of buff are sent from task id root to all other tasks. Could be done by putting MPI_Send in a loop.

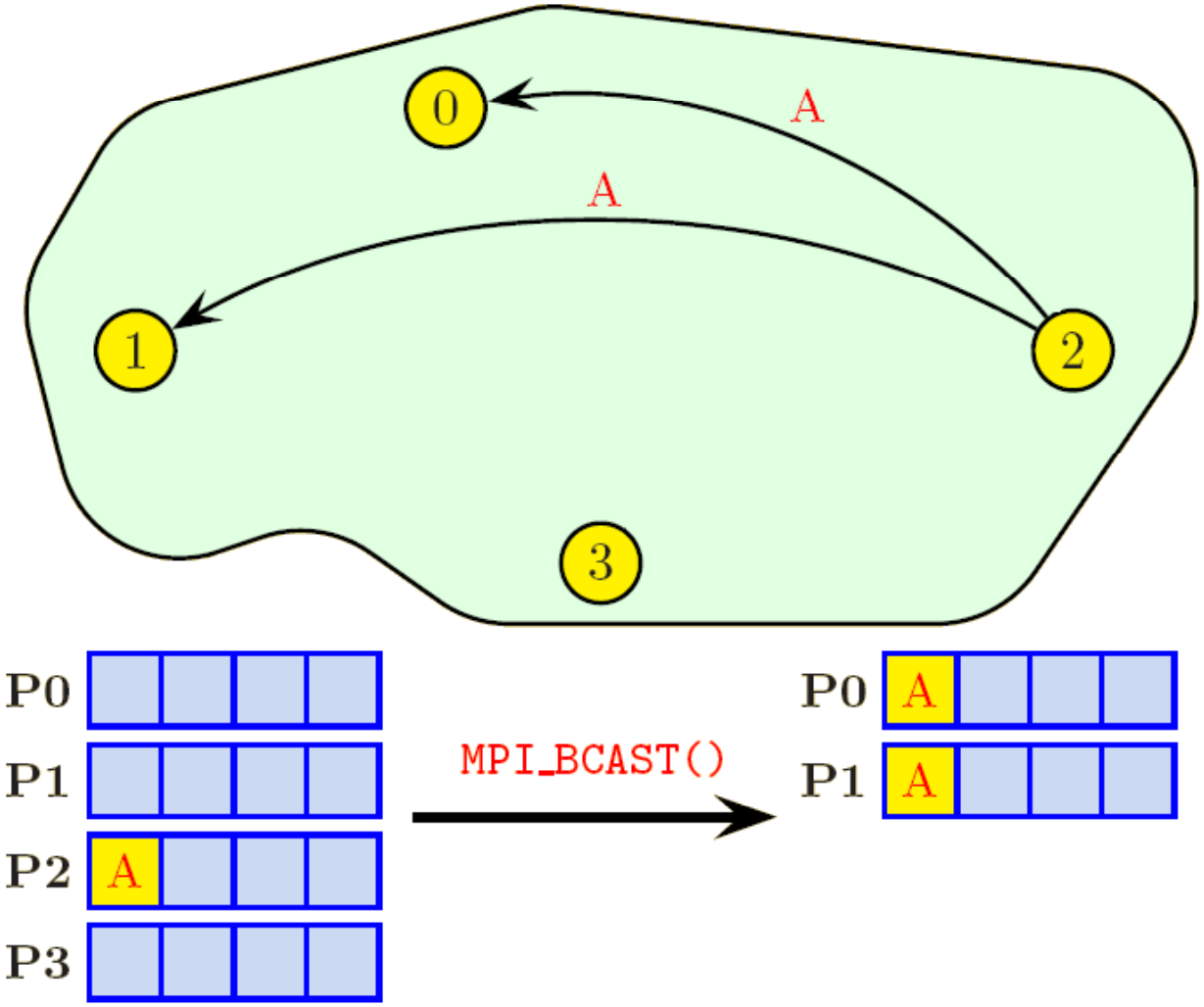
MPI_BROADCAST



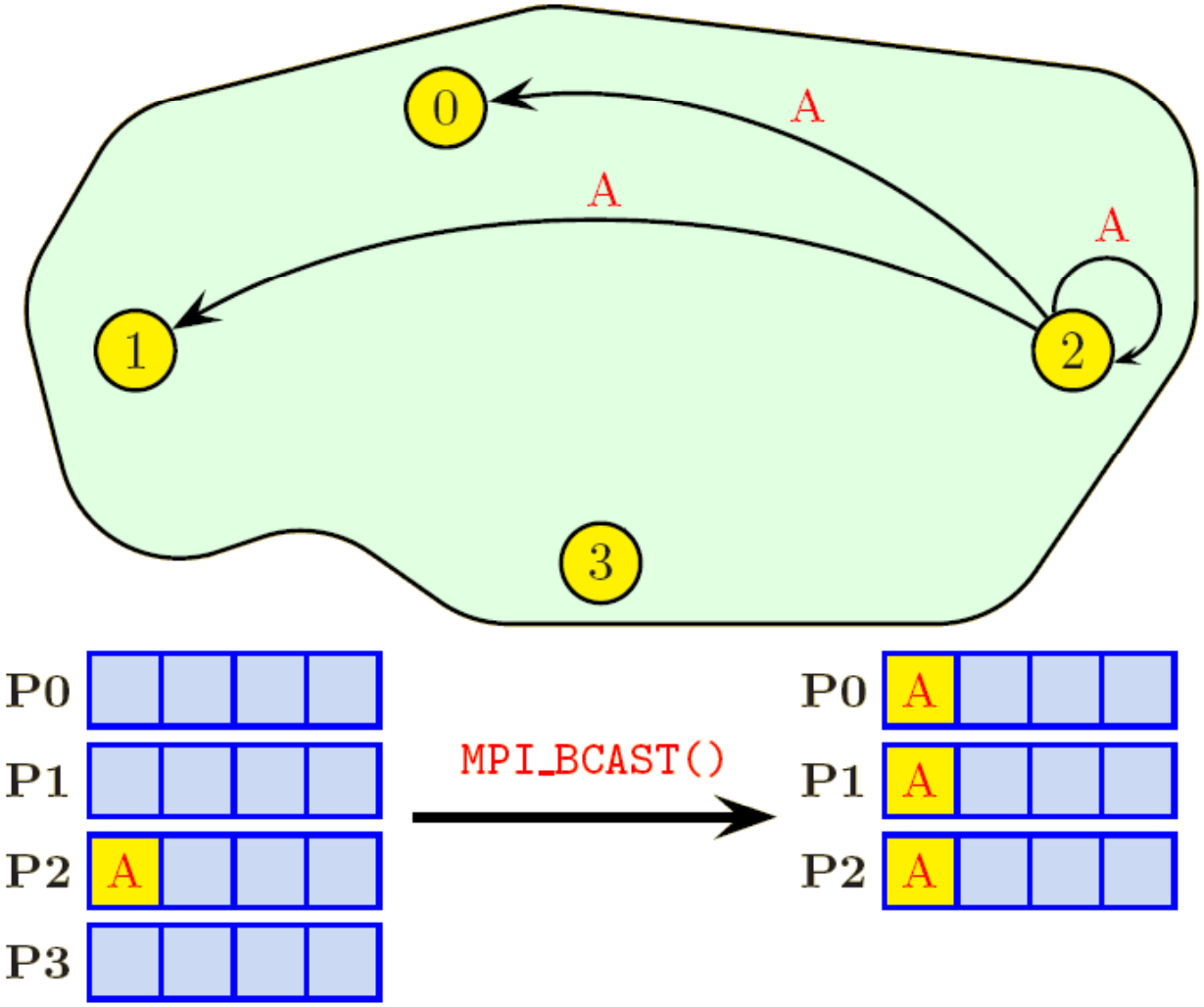
MPI_BROADCAST



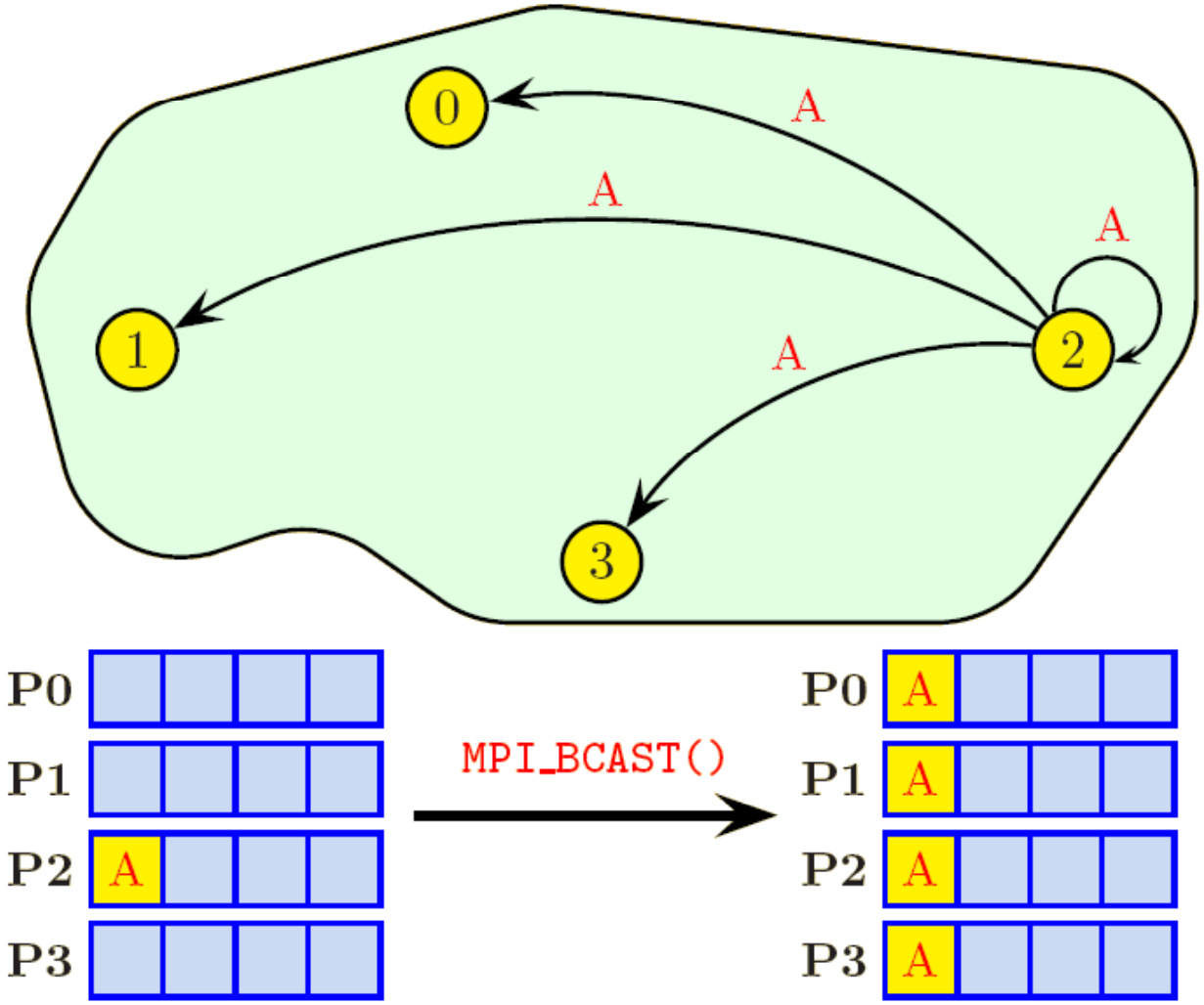
MPI_BROADCAST



MPI_BROADCAST



MPI_BROADCAST



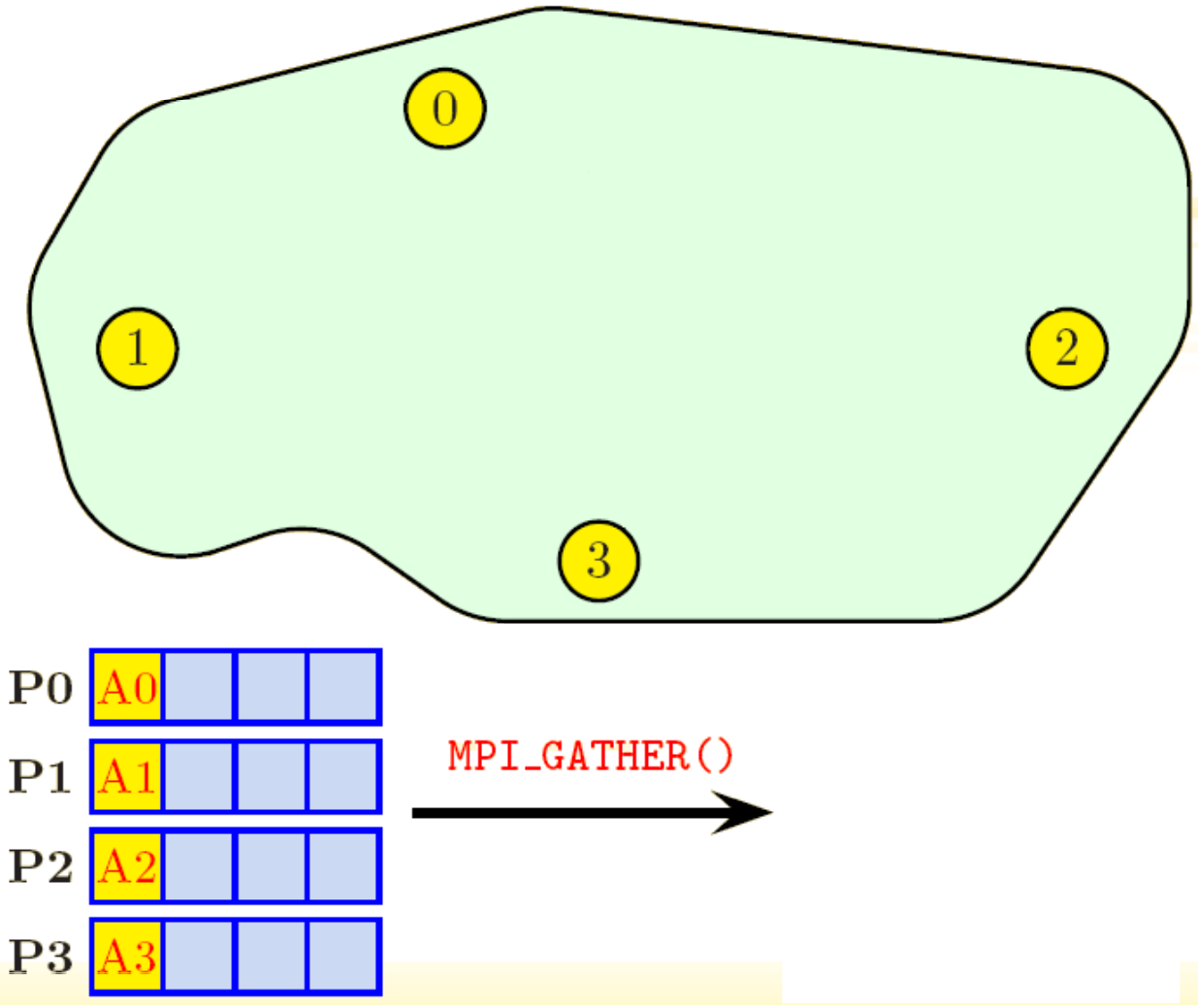
MPI_GATHER

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

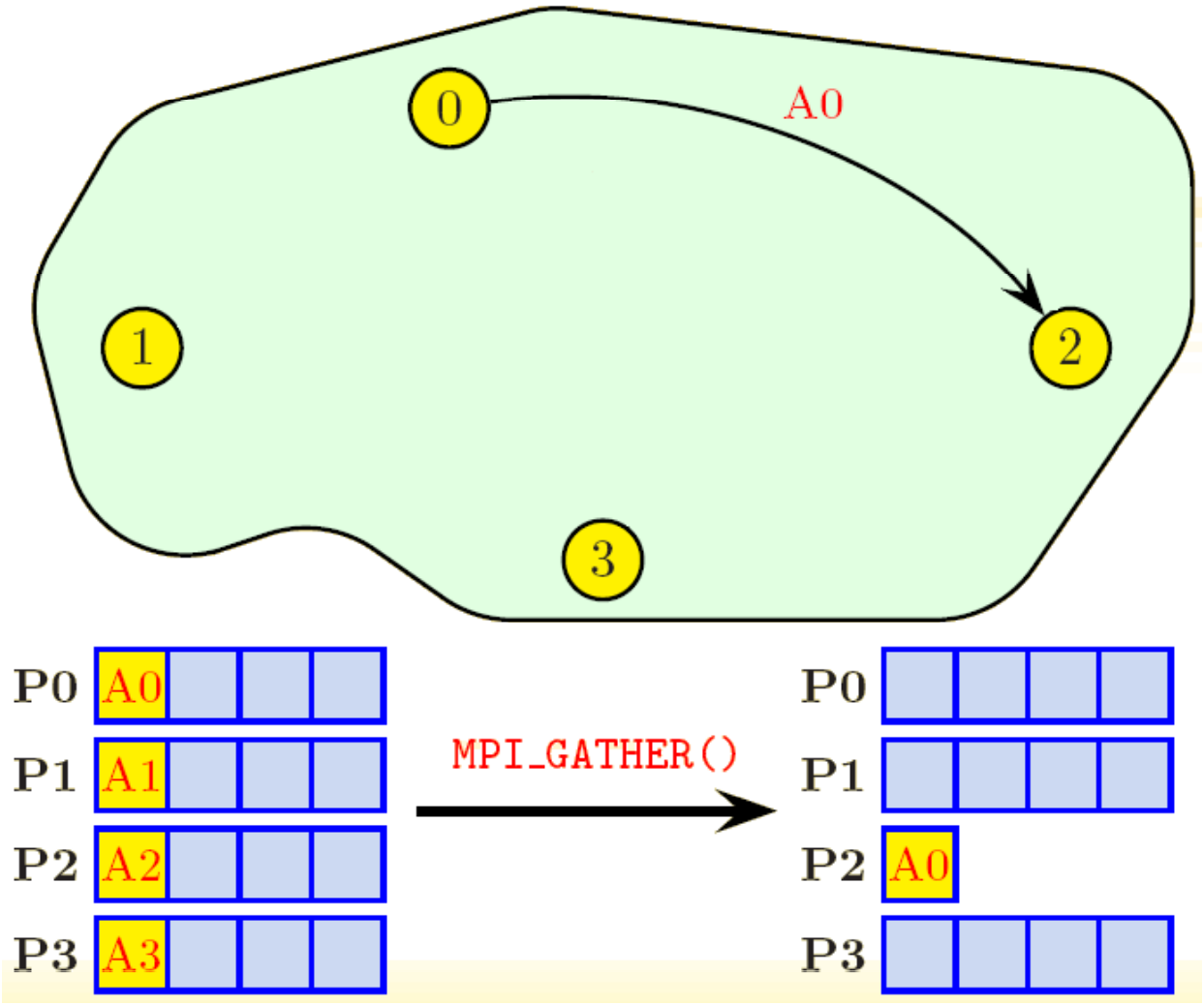
ROOT	**task doing gather**	**input**
SBUFF	**array being sent**	**input**
RBUFF	**array being received**	**output**
COUNT	**number of items from each task**	**input**

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

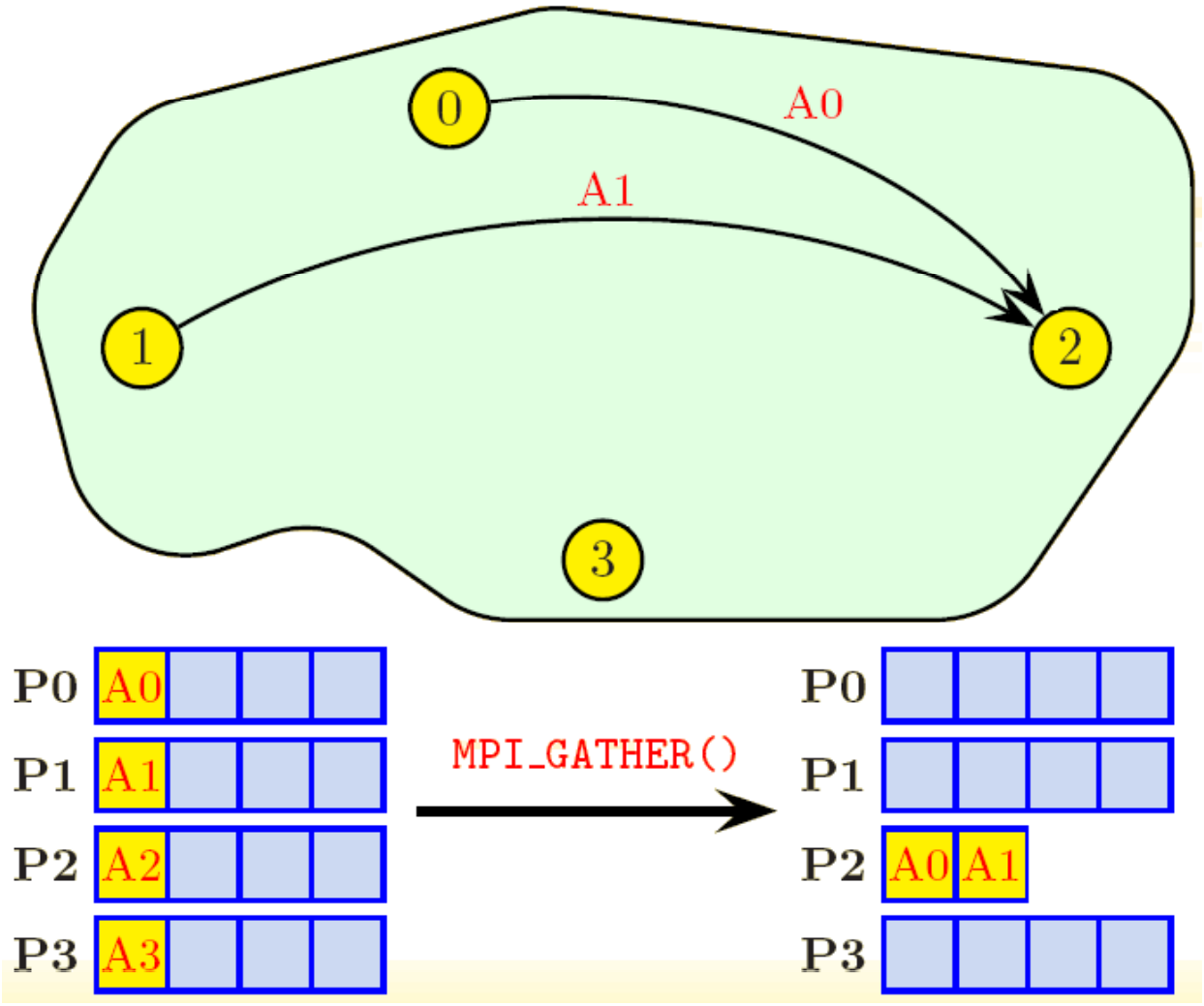
MPI_GATHER



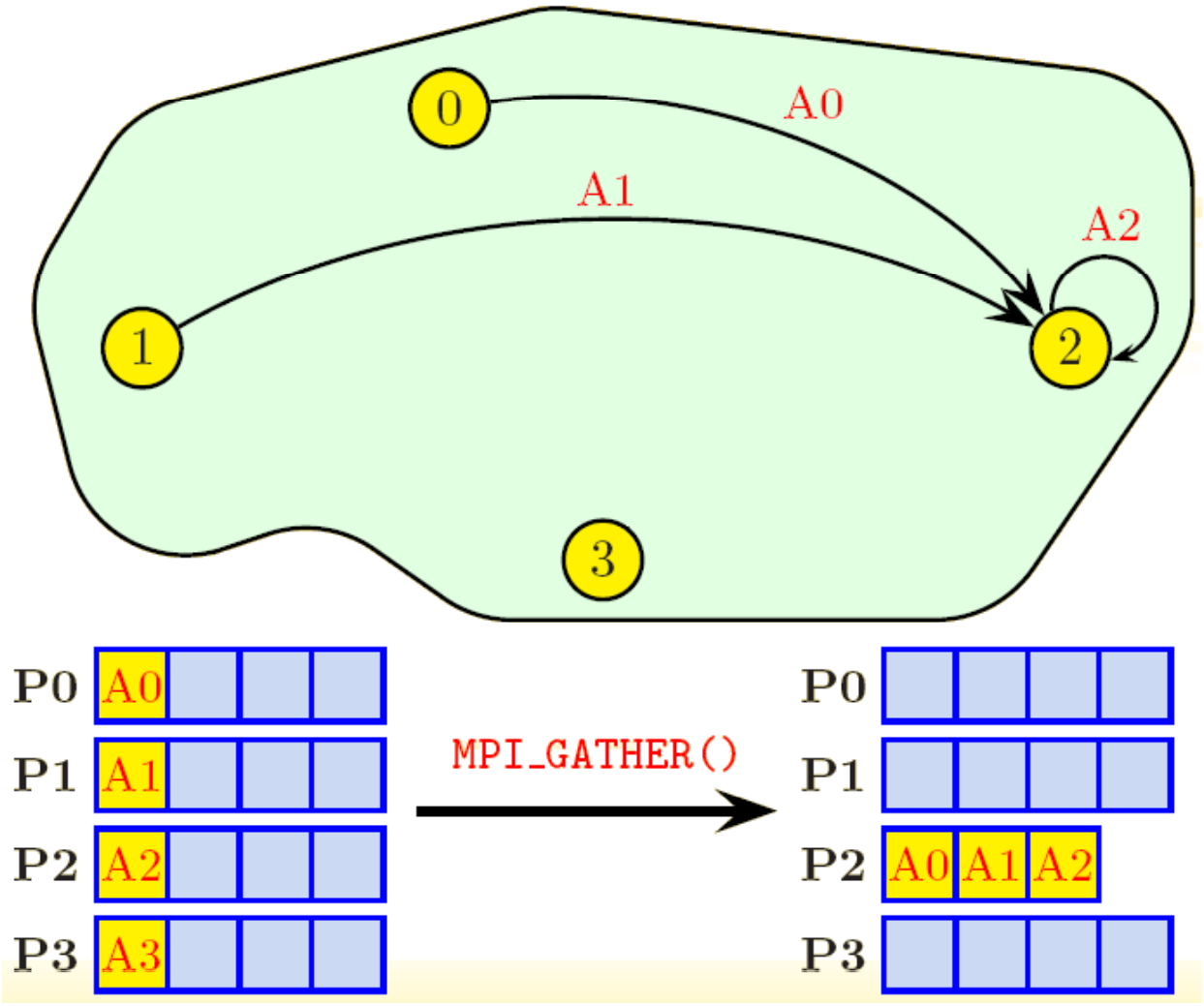
MPI_GATHER



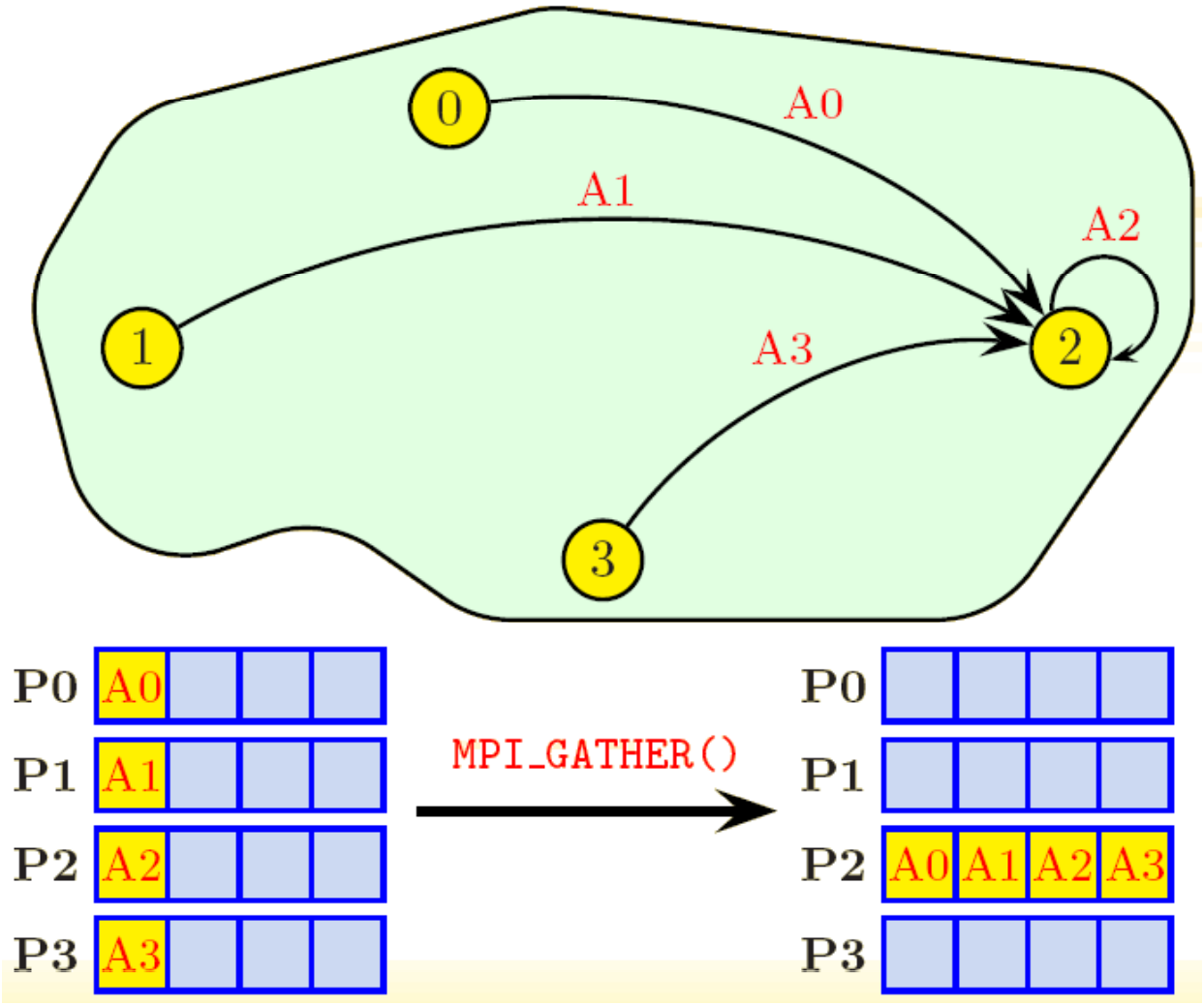
MPI_GATHER



MPI_GATHER



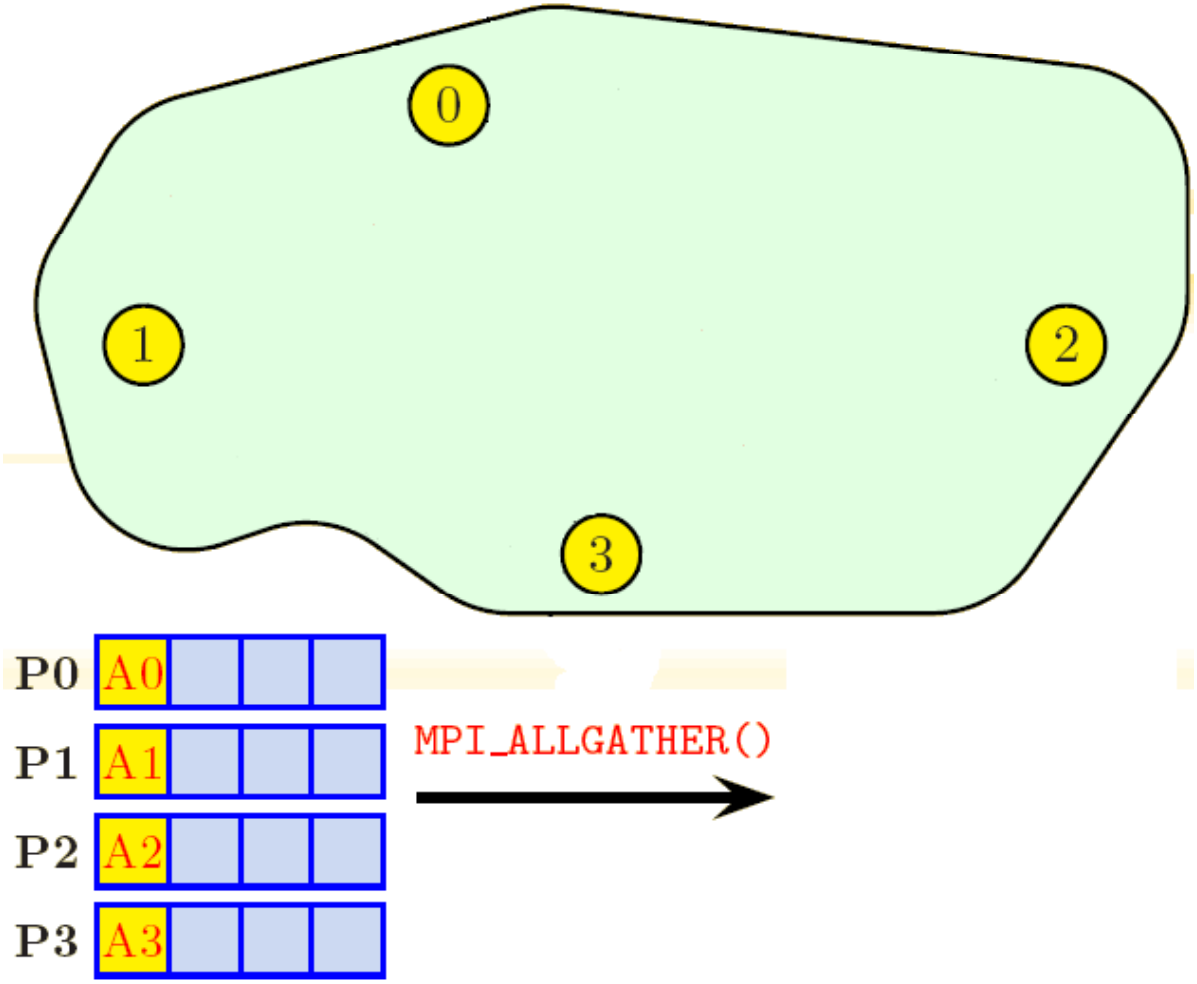
MPI_GATHER


0
A0
A1
A2
1
2
A3
3
P0
A0
P1
A1
P2
A2
P3
A3
MPI_GATHER()
P0
P1
P2
A0
A1
A2
A3
P3

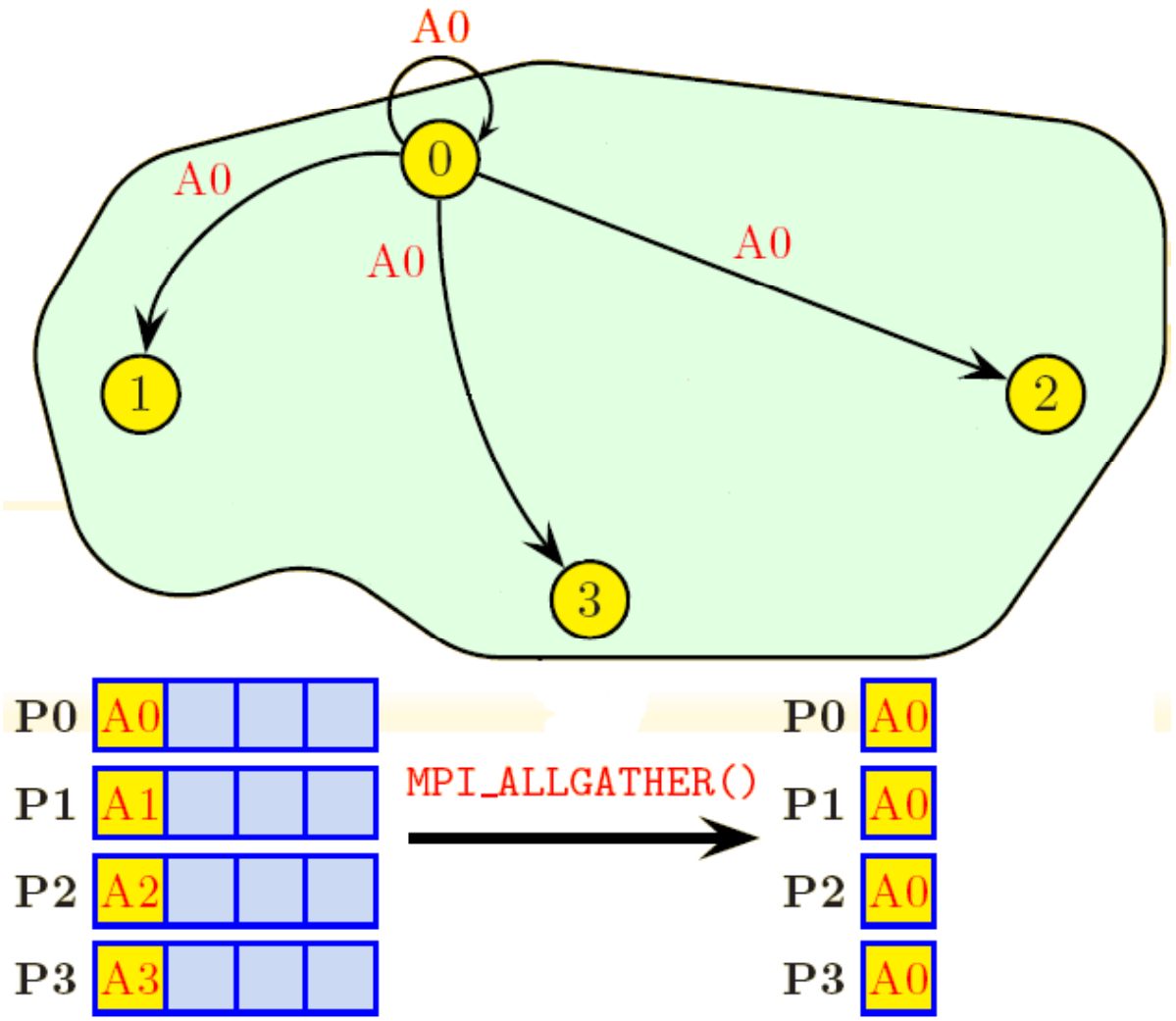
Gather Routines

- **MPI_ALLGATHER**
  - **gather arrays of equal length into one array on all tasks**
- **MPI_GATHERV**
  - **gather arrays of different lengths into one array on one task**
- **MPI_ALLGATHERV**
  - **gather arrays of different lengths into one array on all tasks**

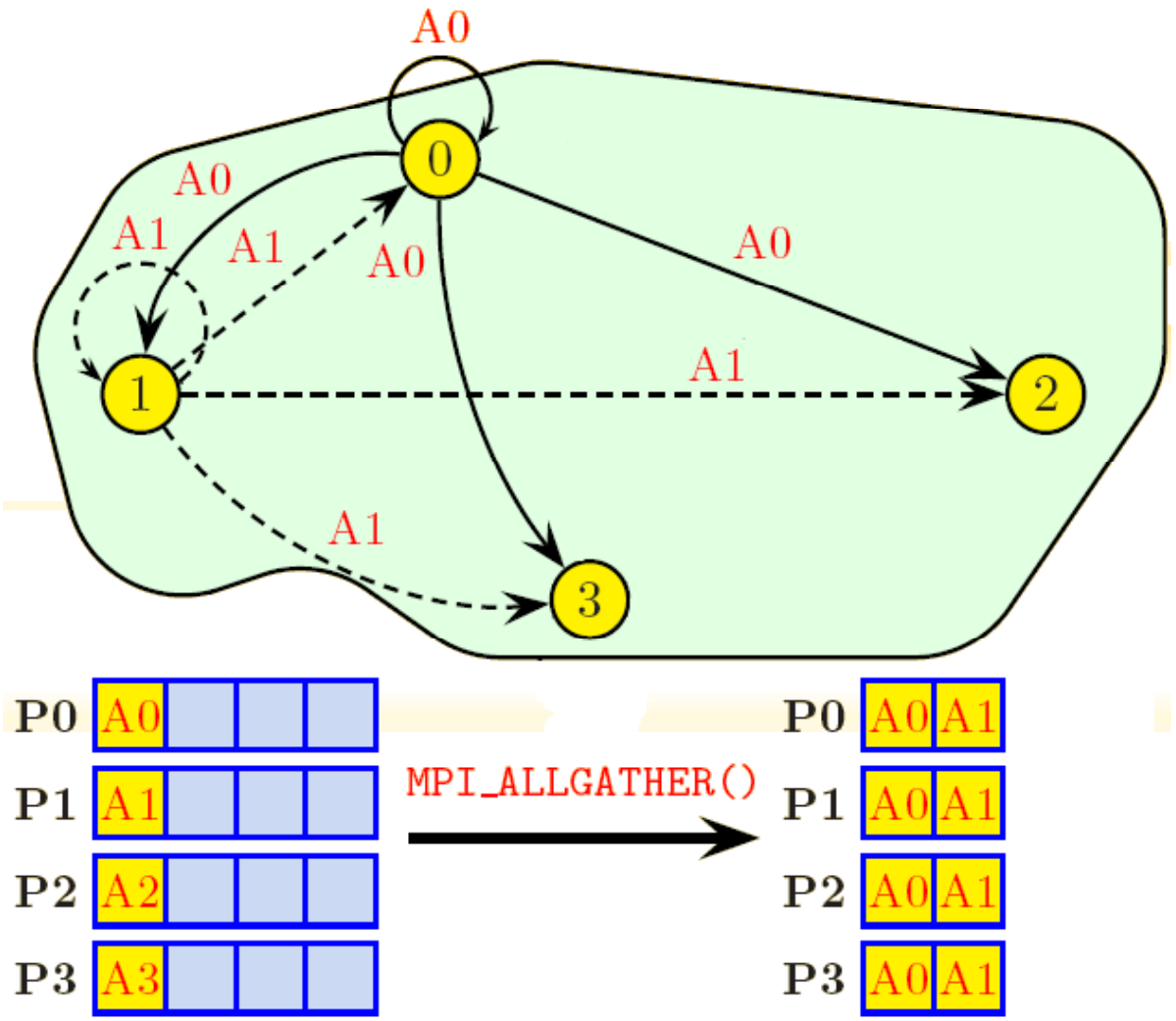
MPI_ALLGATHER



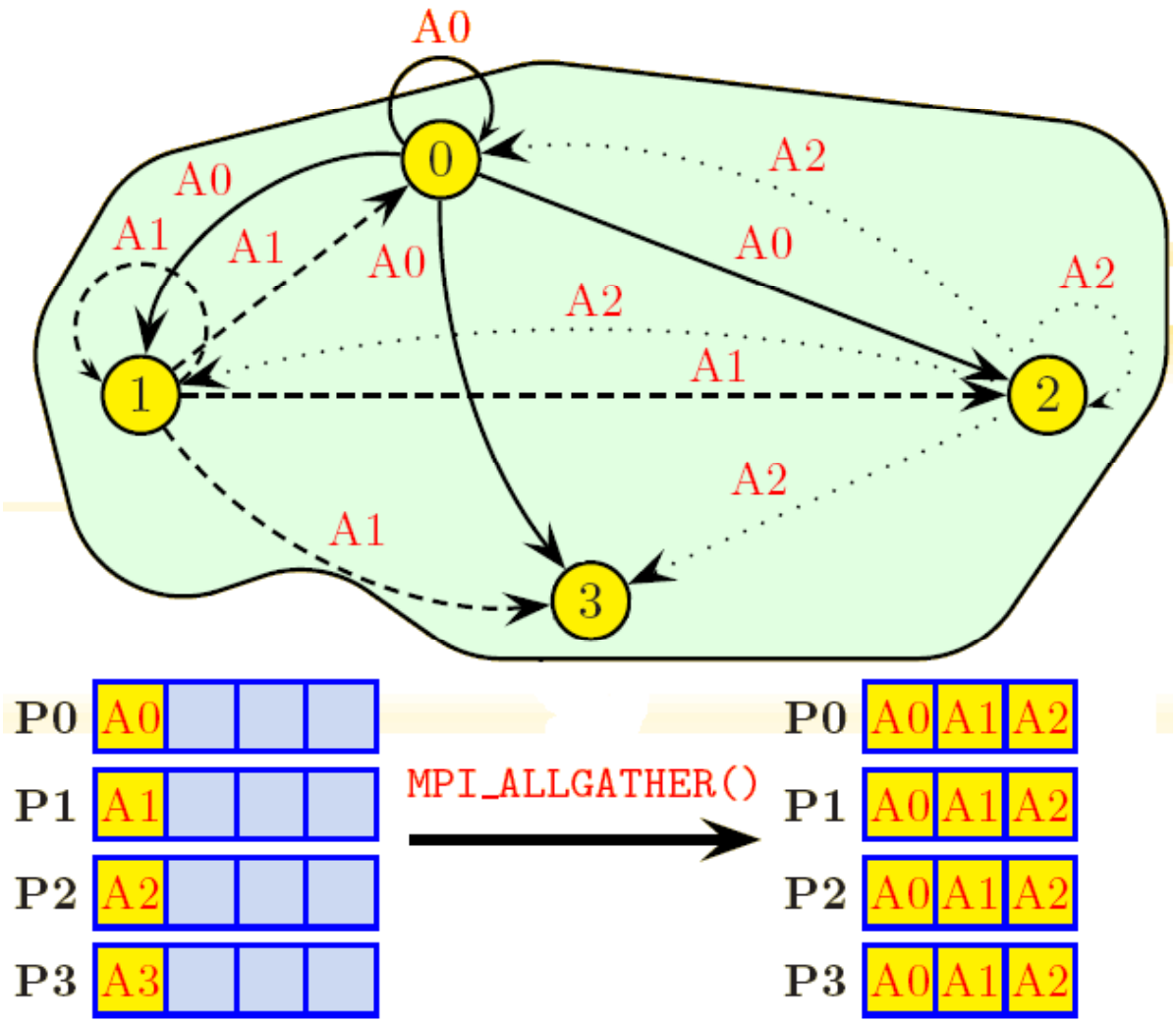
MPI_ALLGATHER



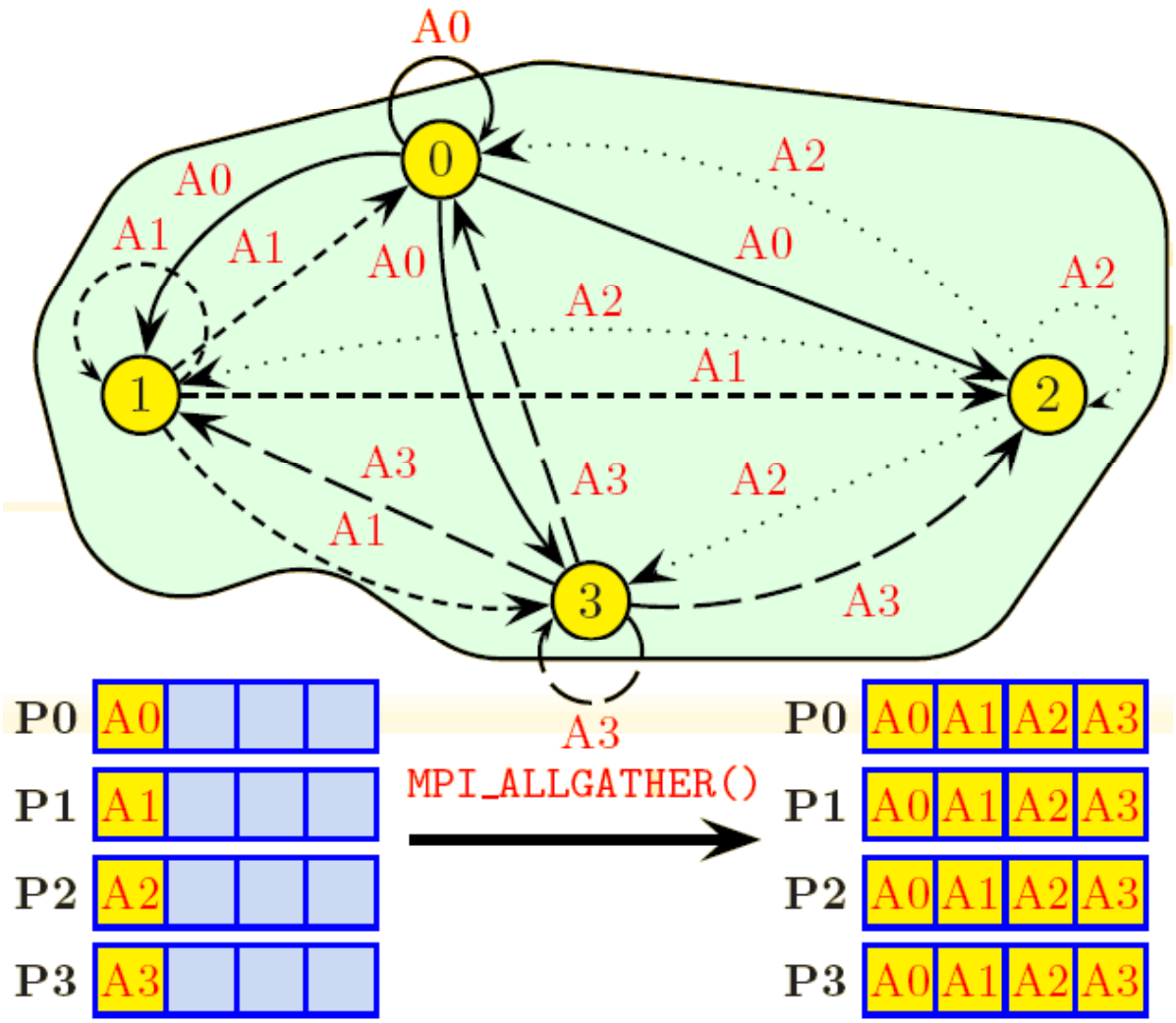
MPI_ALLGATHER



MPI_ALLGATHER



MPI_ALLGATHER

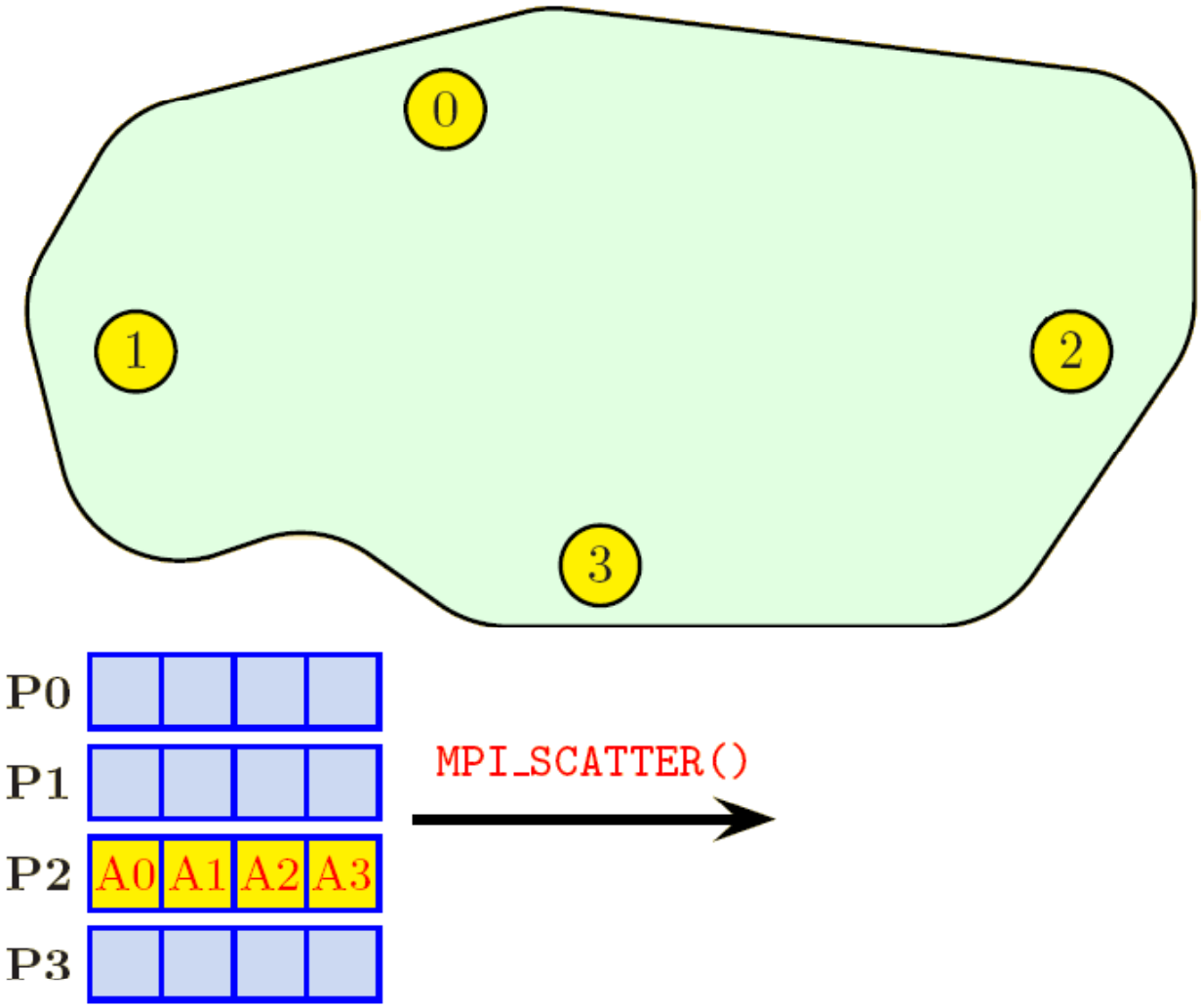


Scatter Routines

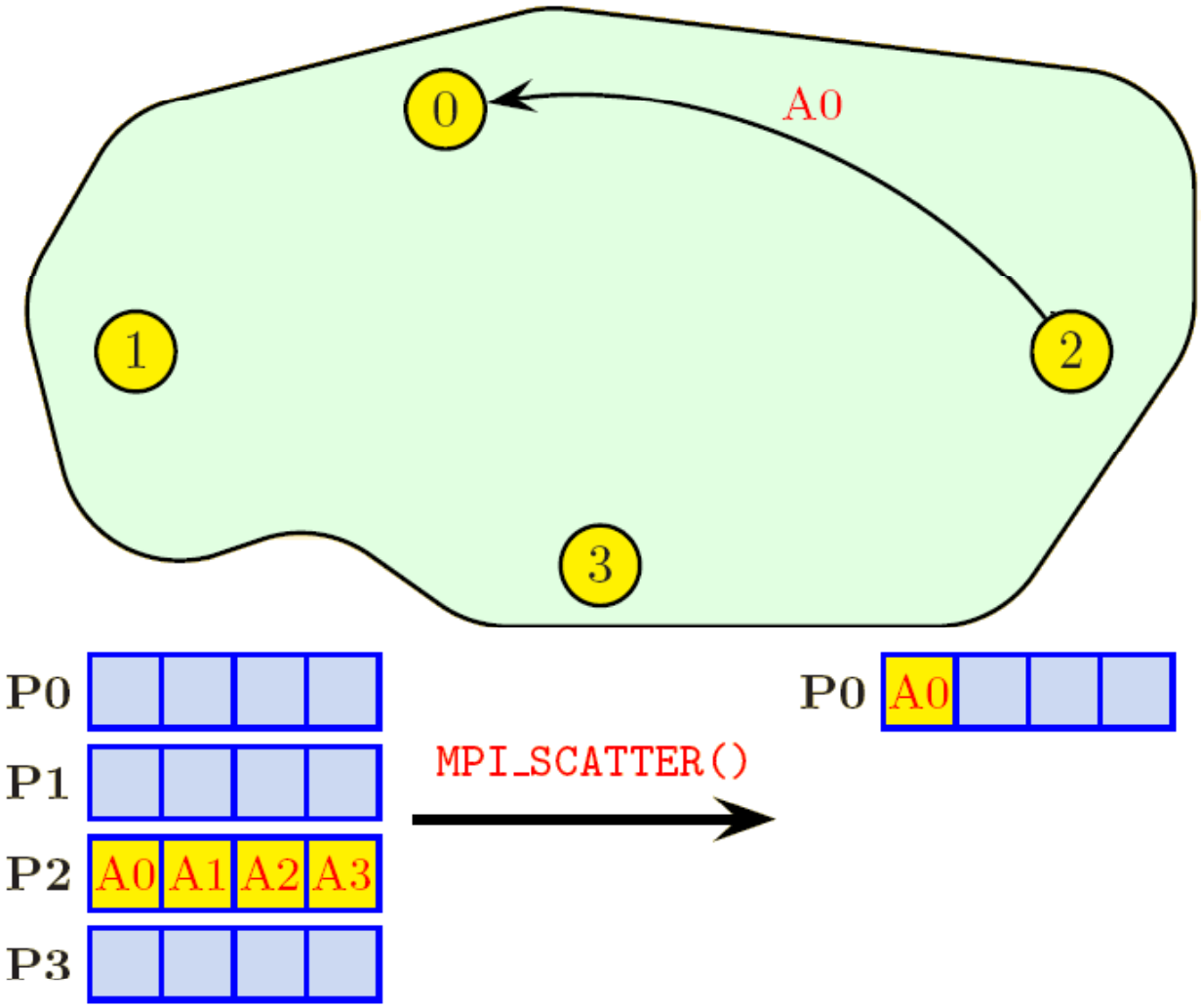
- **MPI_SCATTER**
  - **divide one array on one task equally amongst all tasks**

- **MPI_SCATTERV**
  - **divide one array on one task unequally amongst all tasks**

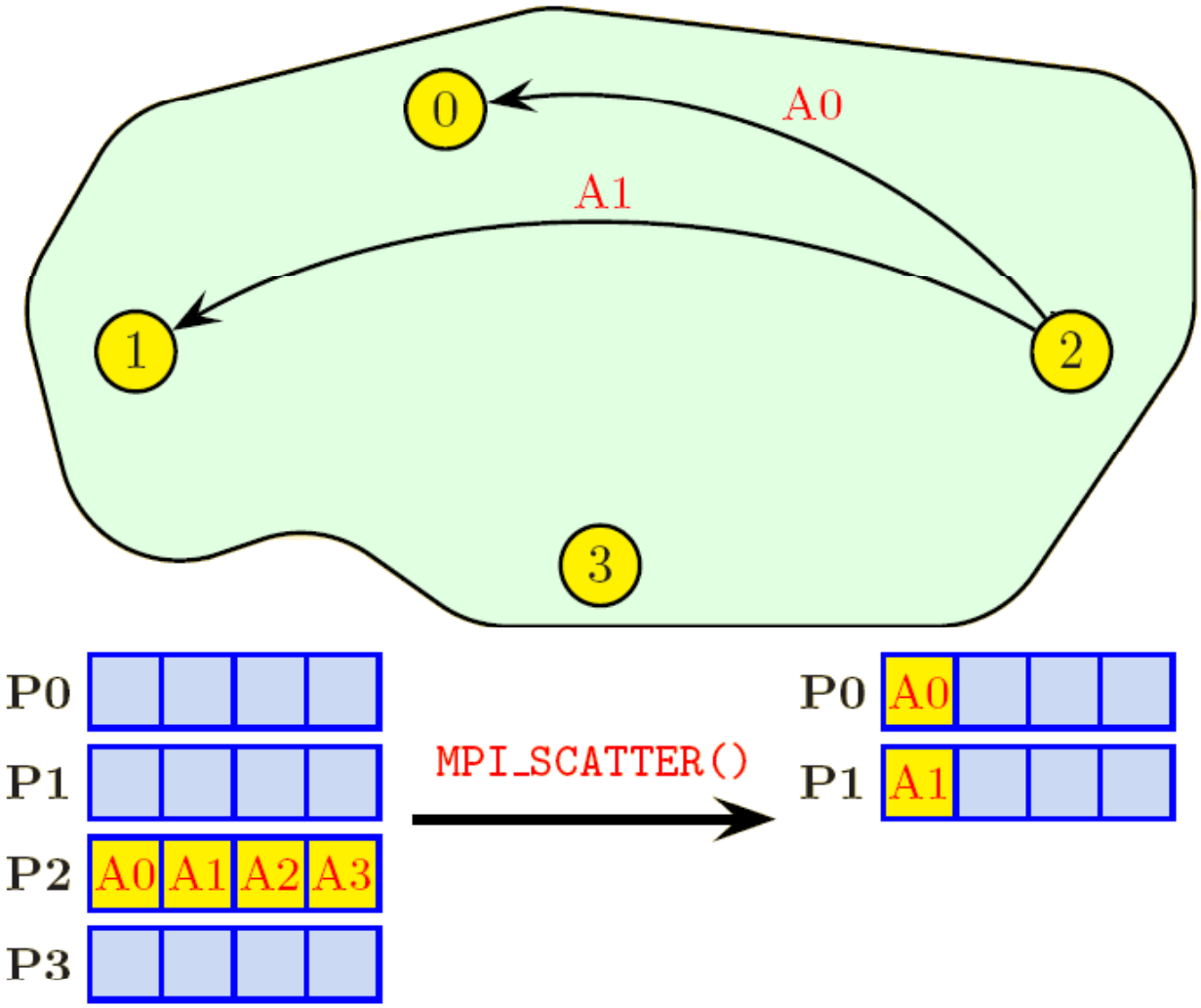
MPI_SCATTER



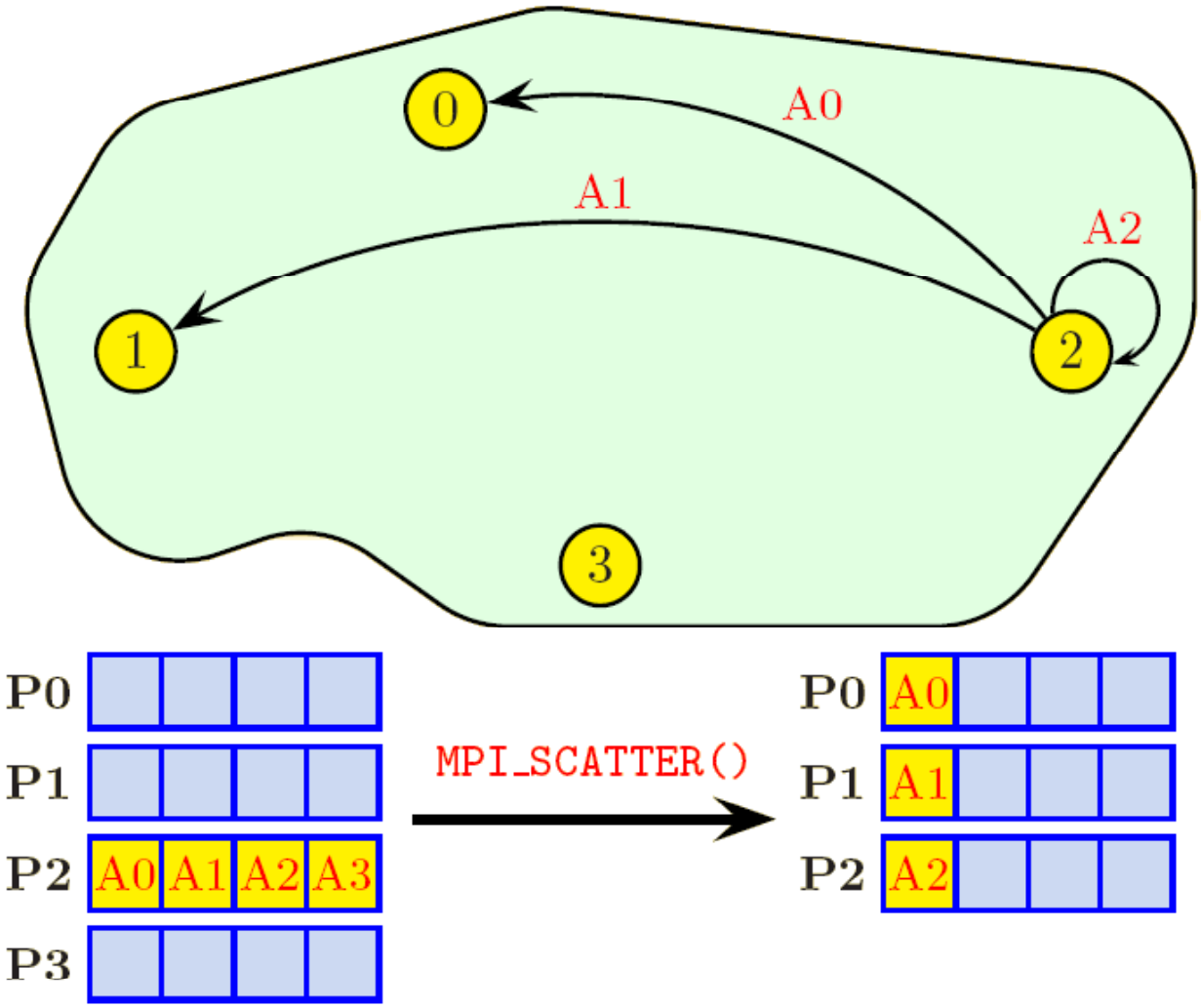
MPI_SCATTER



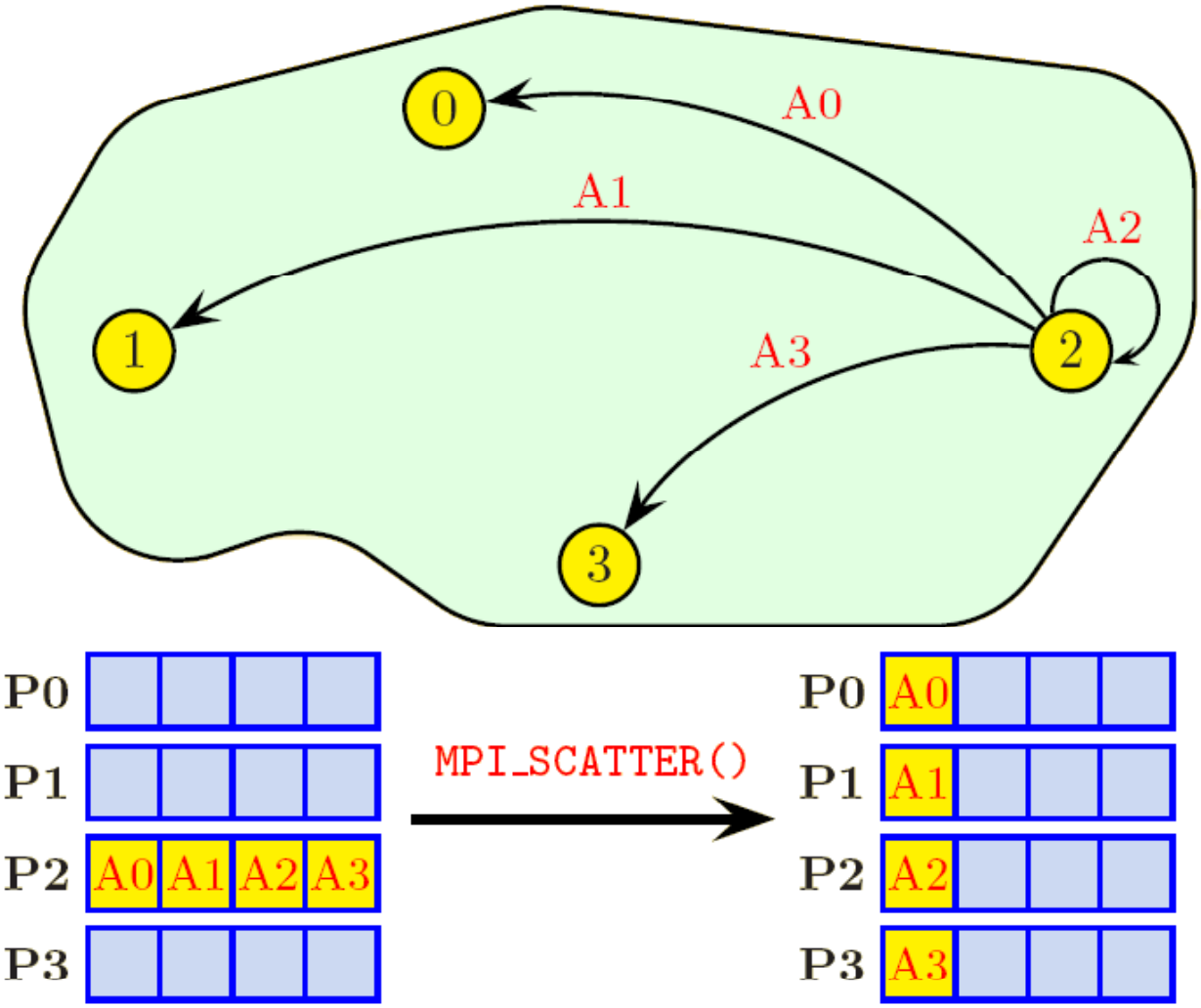
MPI_SCATTER



MPI_SCATTER



MPI_SCATTER

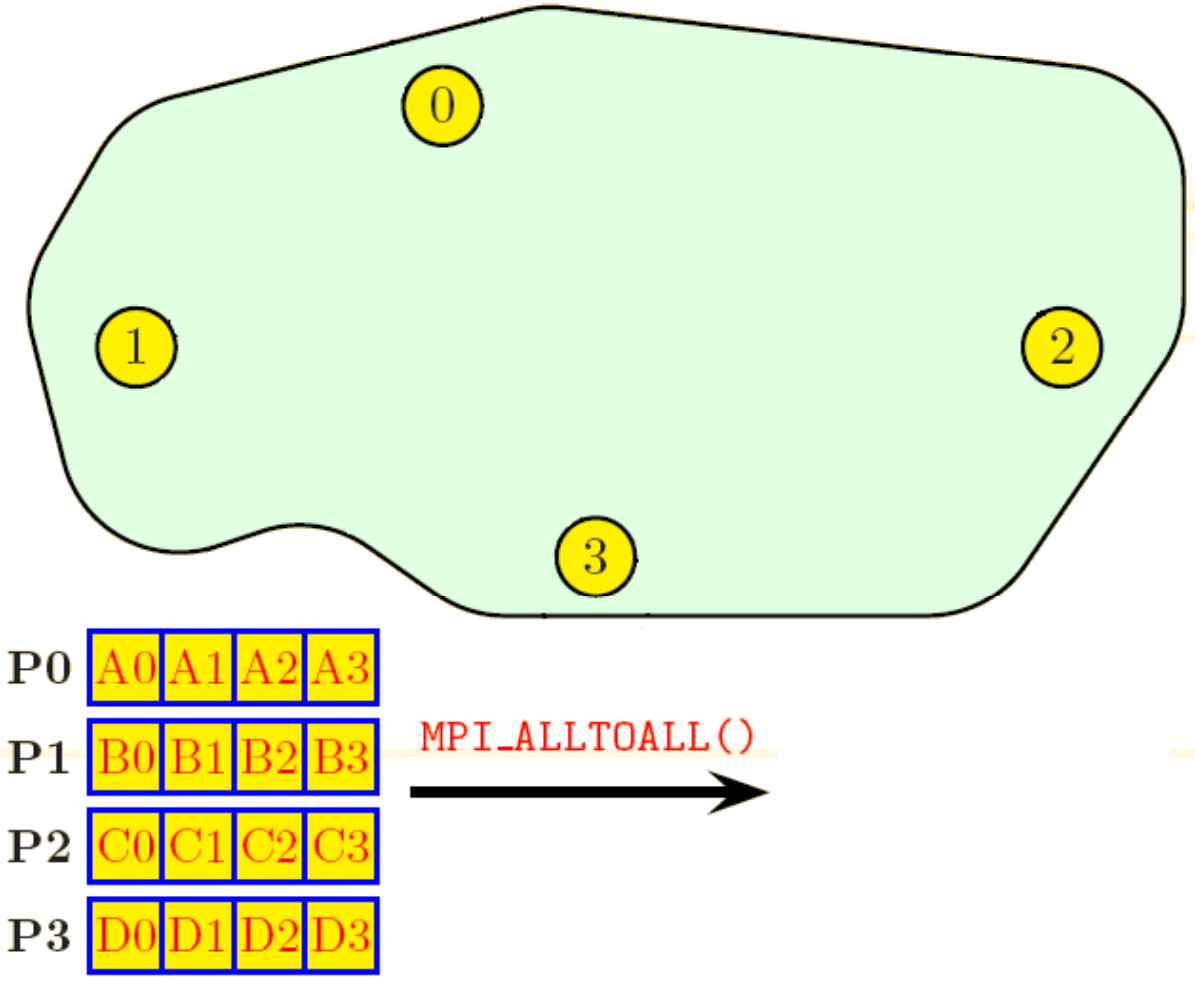

0
A0
A1
A2
1
2
A3
3
P0
P1
P2
P3
A0
A1
A2
A3
MPI_SCATTER()
P0
P1
P2
P3
A0
A1
A2
A3

All to All Routines

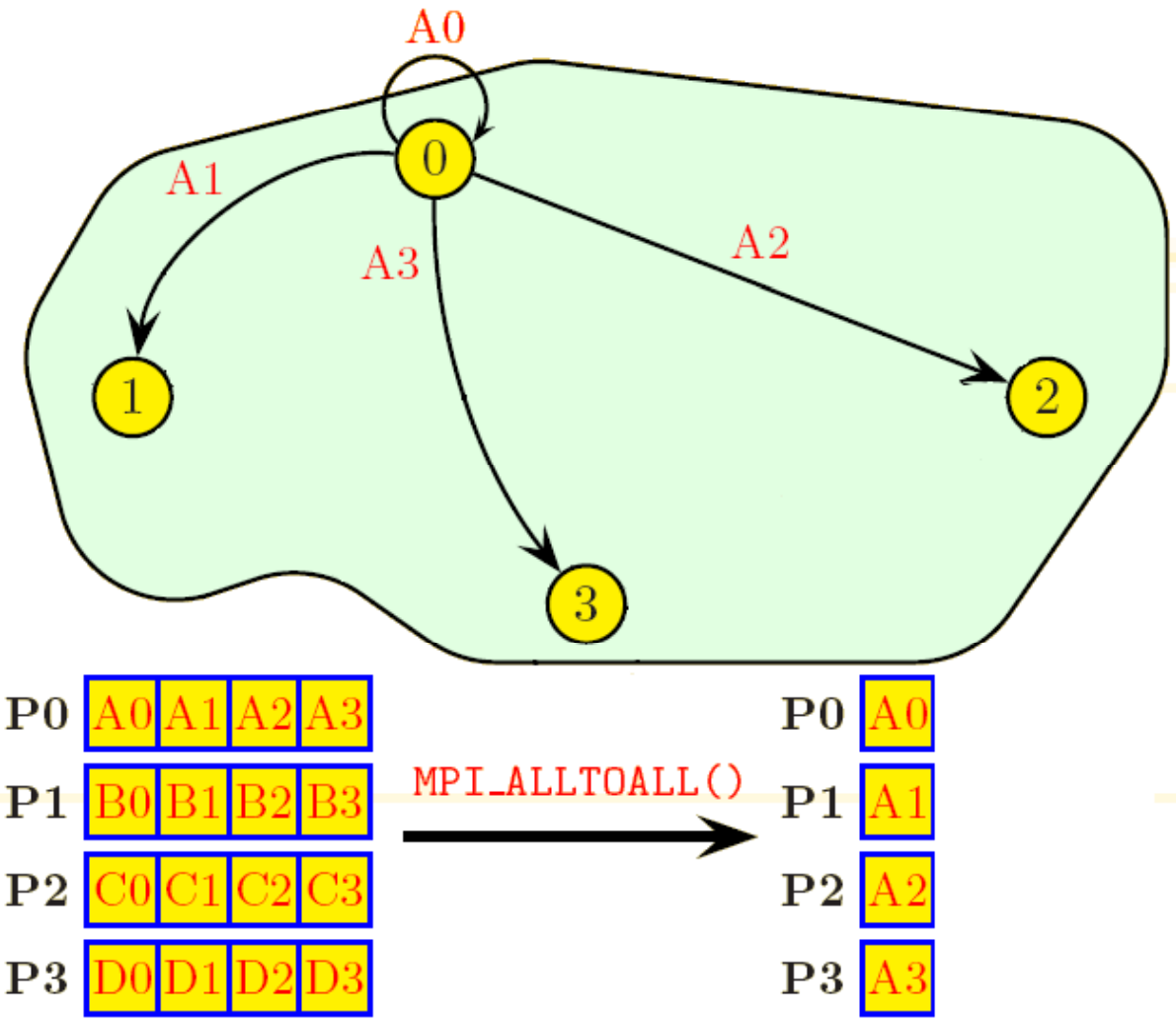
- **MPI_ALLTOALL**
  - **every task sends equal length parts of an array to all other tasks**
  - **every task receives equal parts from all other tasks**

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

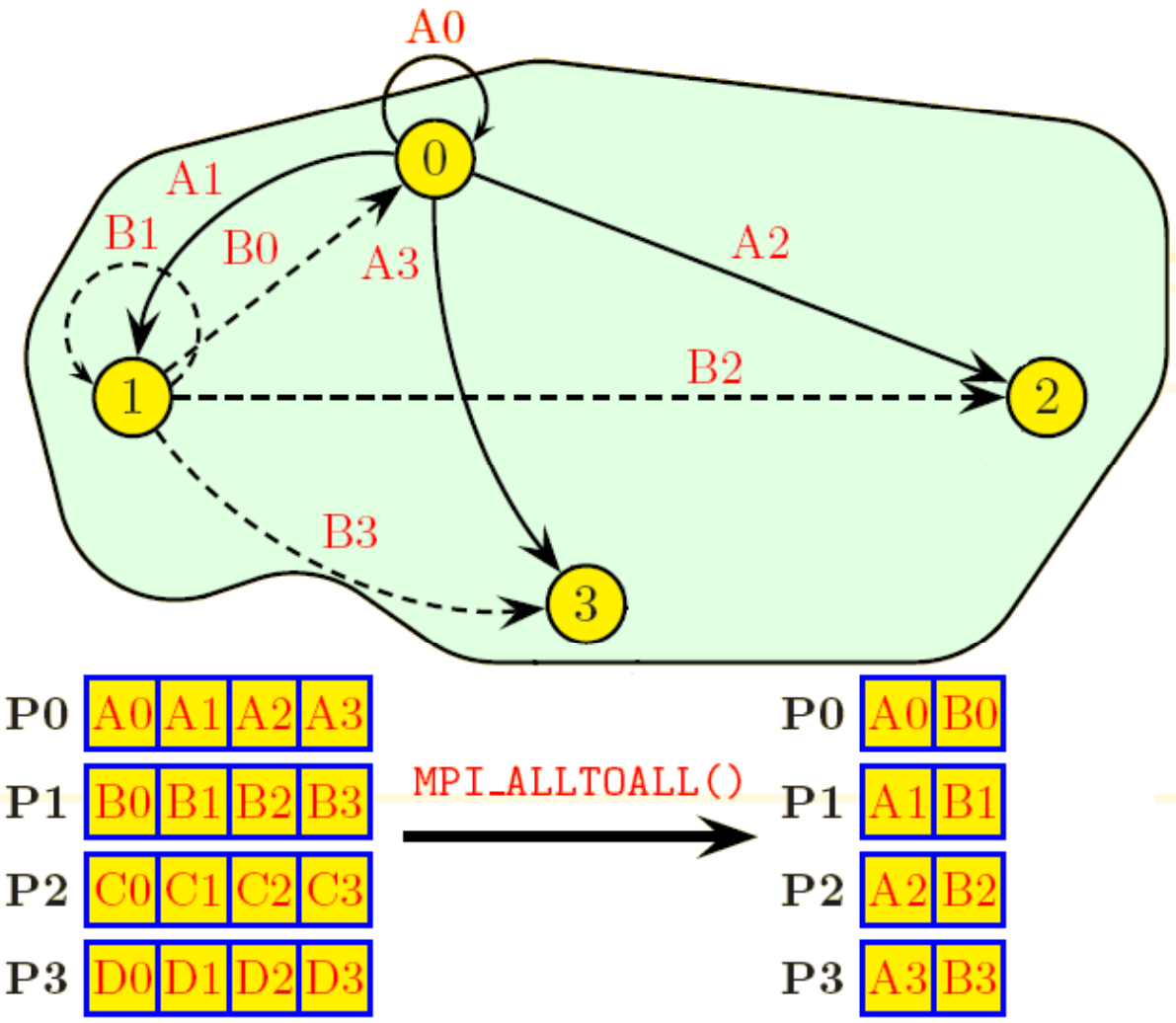
MPI_ALLTOALL



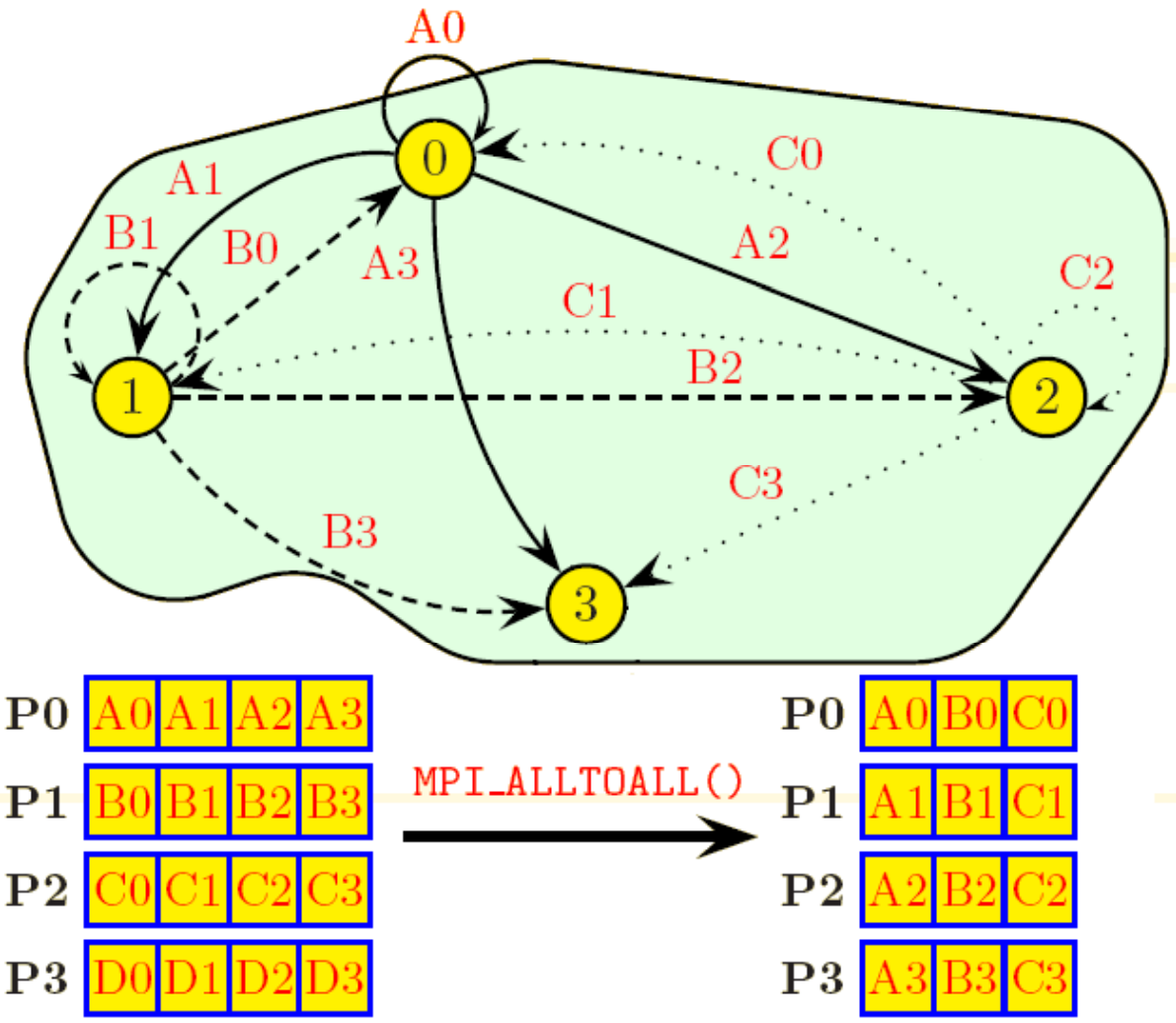
MPI_ALLTOALL



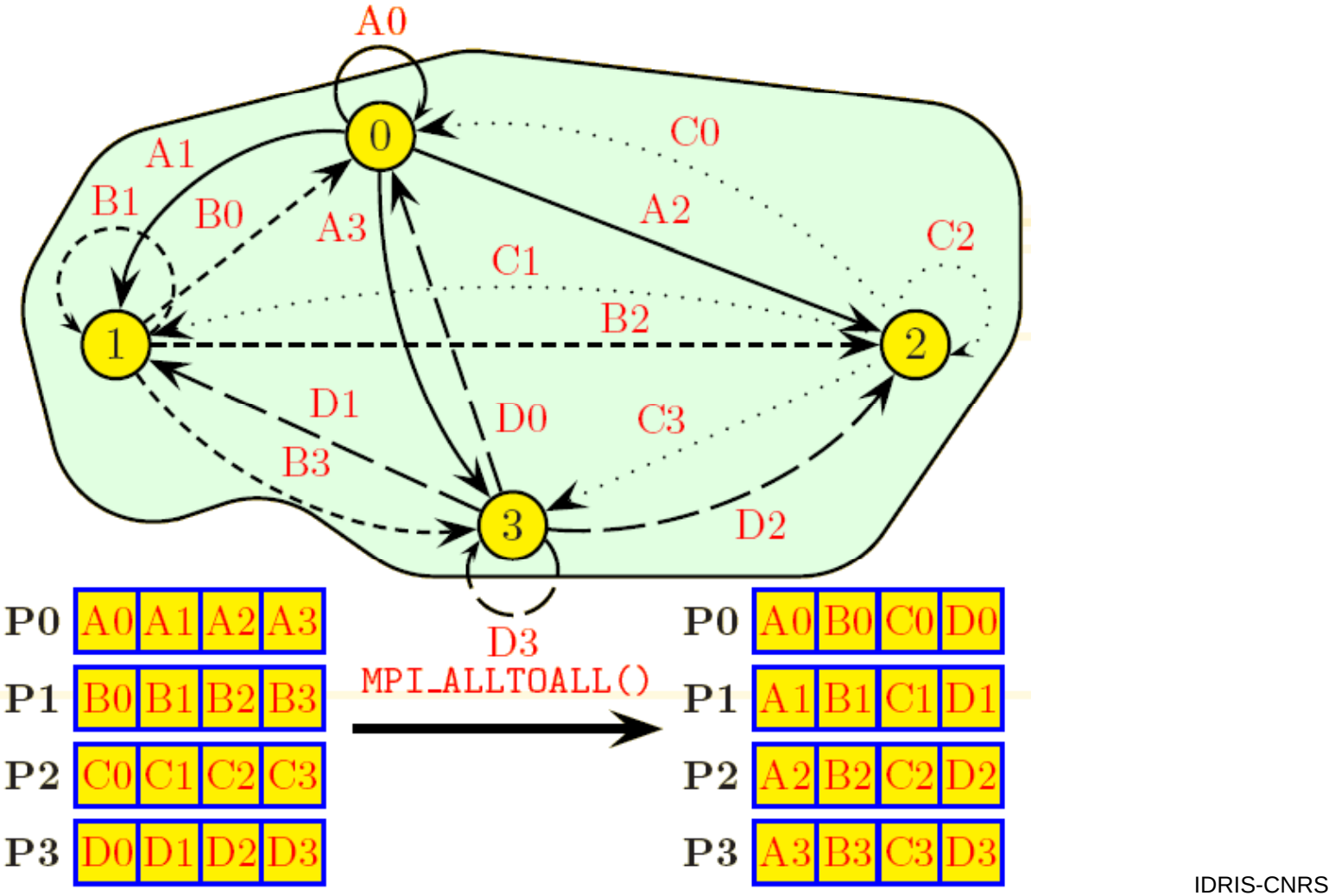
MPI_ALLTOALL



MPI_ALLTOALL



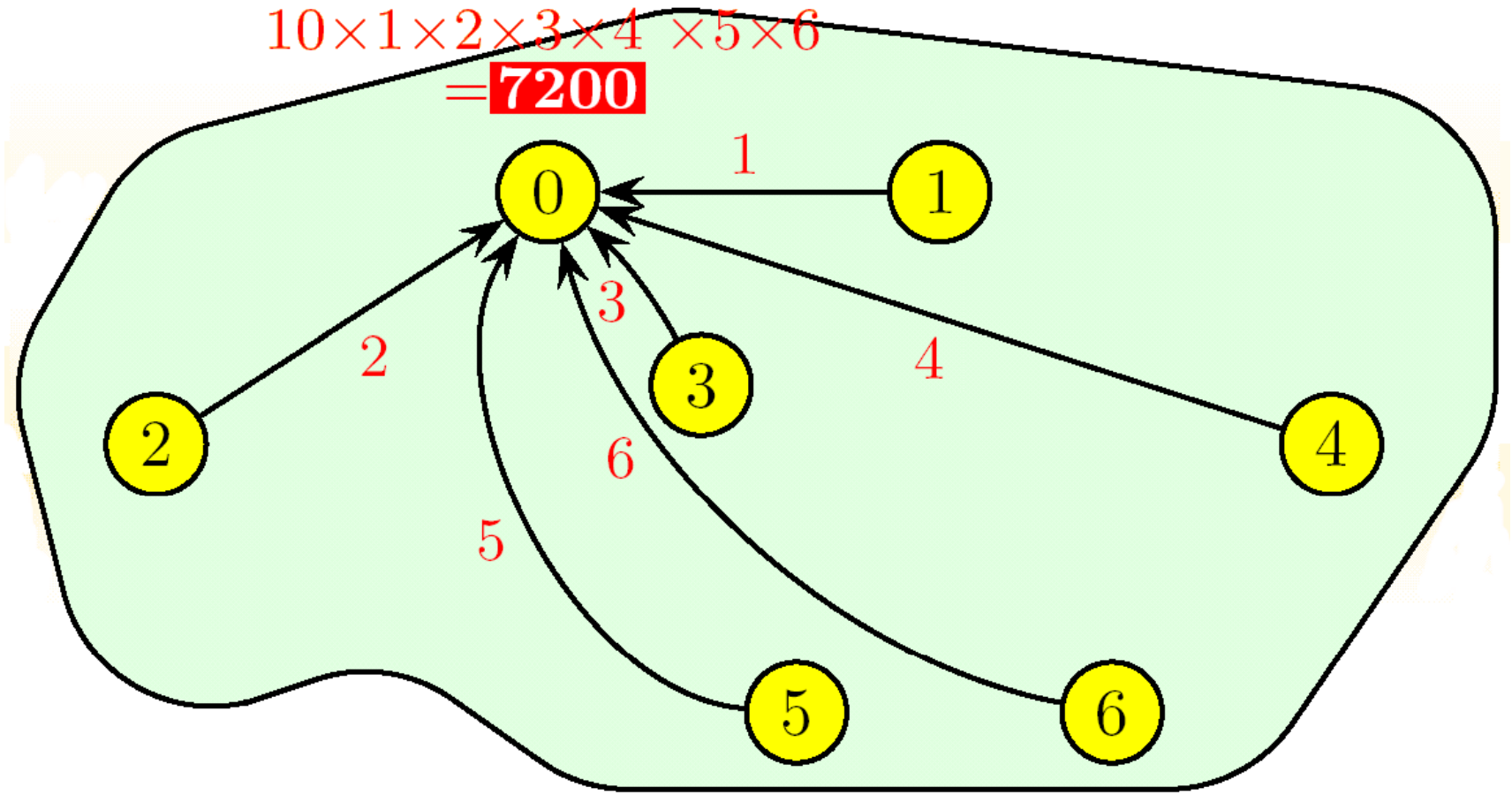
MPI_ALLTOALL



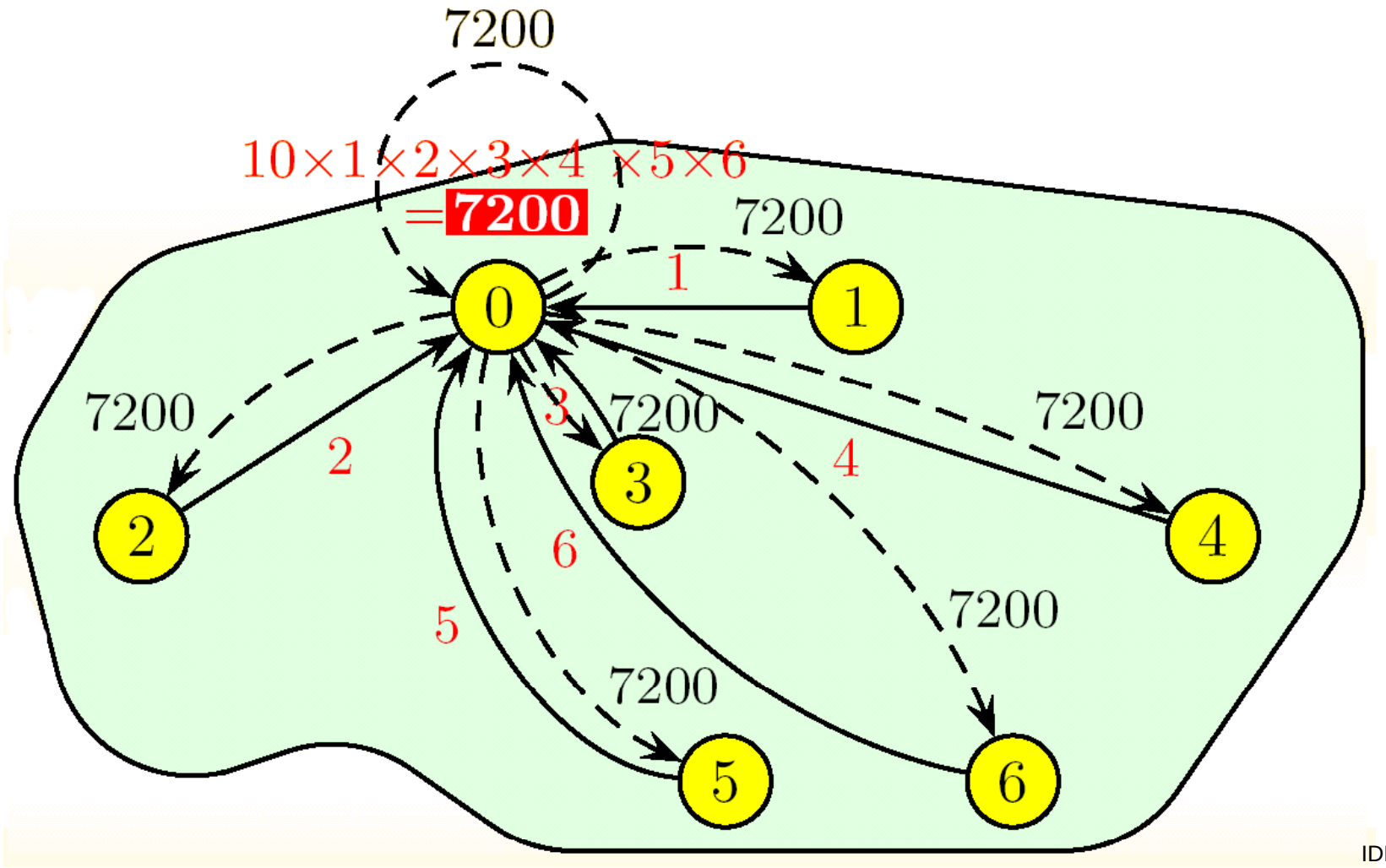
Reduction routines

- **Do 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 and result is computed on one task**
- **MPI_ALLREDUCE**
  - **every task sends, result is computed and broadcast to all**

MPI_REDUCE



MPI_ALLREDUCE



IDRIS-CNRS

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**

  `www-unix.mcs.anl.gov/mpi/index.html`

- **IBM Parallel Environment for AIX Manuals**

  `www.ibm.com/servers/eserver/pseries/library/sp_books/pe.html`

  - IBM PE Hitchhikers Guide (sample programs also available)
  - MPI Programming Guide
  - MPI Subroutine Reference

- **Further Training Material**

  `www.epcc.ed.ac.uk/computing/training/document_archive/`

  - Decomposing the Potentially Parallel
  - MPI Course

Second Practical

- **exercise1b**
- **See the README for details**