MPI_ABORT

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

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

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
- 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

More on MPI_RECV

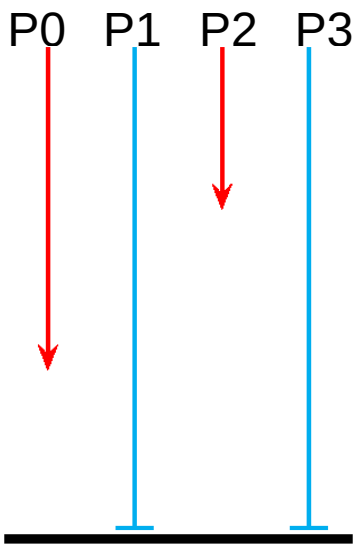
- `MPI_RECV` will block waiting for the message
  - if message never sent then deadlock
    - task will wait until it reaches cpu time limit, and is killed
- The source and tag can be less specific
  - `MPI_ANY_SOURCE` means receive from any sender
  - `MPI_ANY_TAG` means receive any tag
  - Used to 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

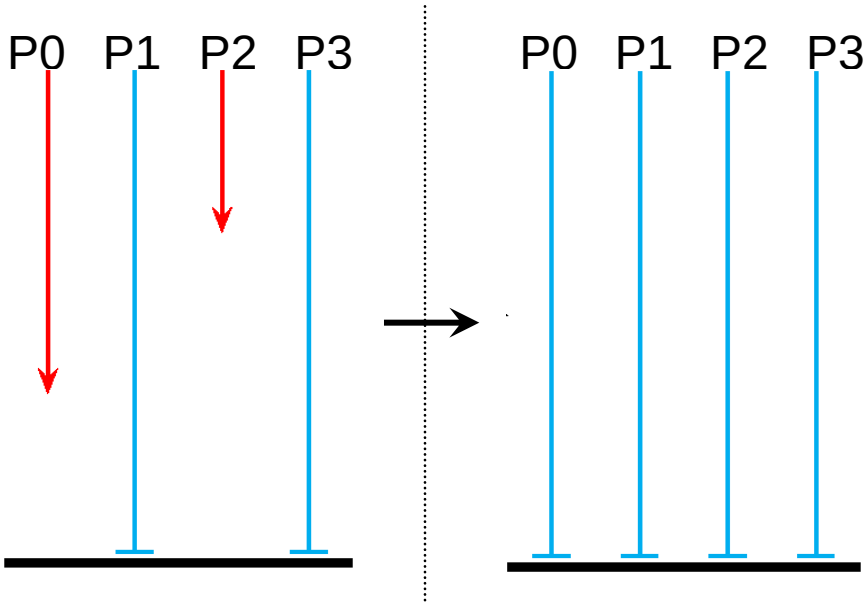
```
integer:: ierror
call MPI_BARRIER(MPI_COMM_WORLD,ierror)
```

- Forces all tasks (in a communicator group) 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
  - `MPI_BARRIER` will wait until the task reaches its cpu limit

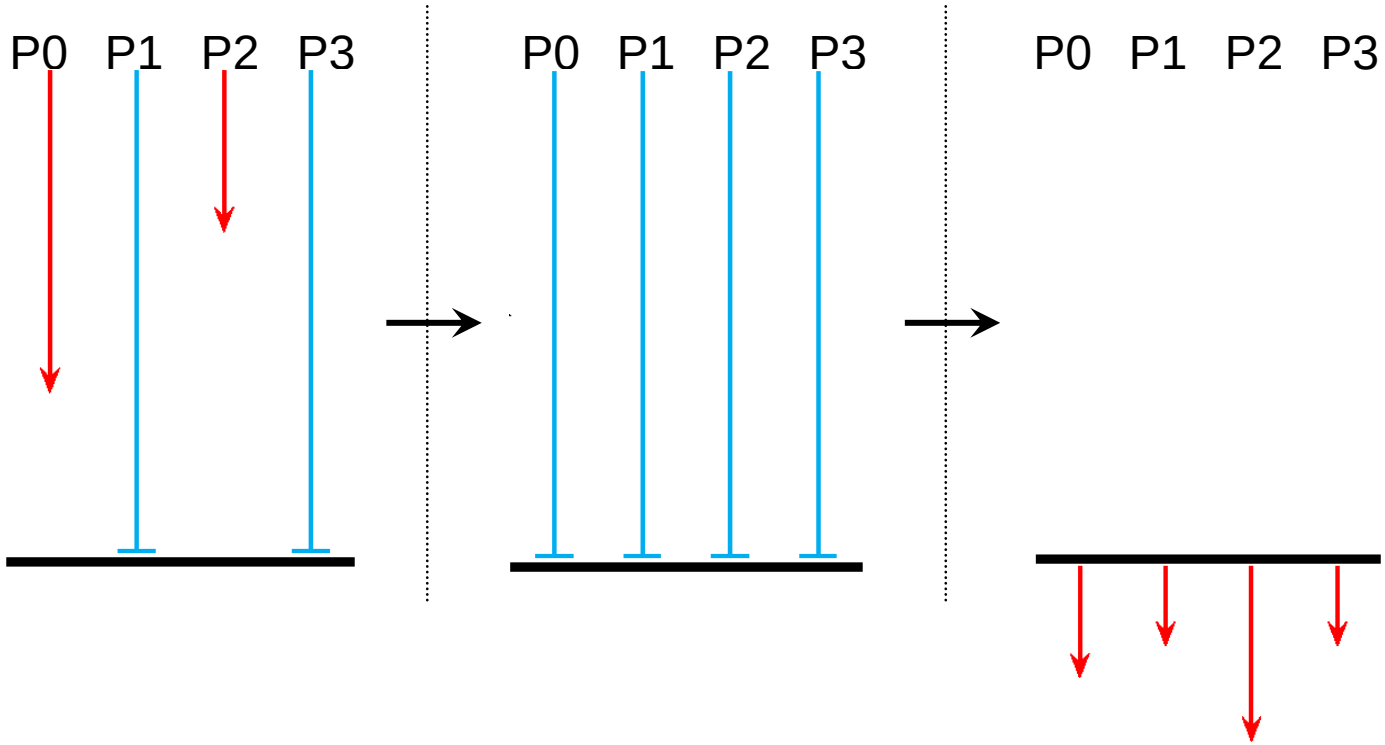
MPI_BARRIER



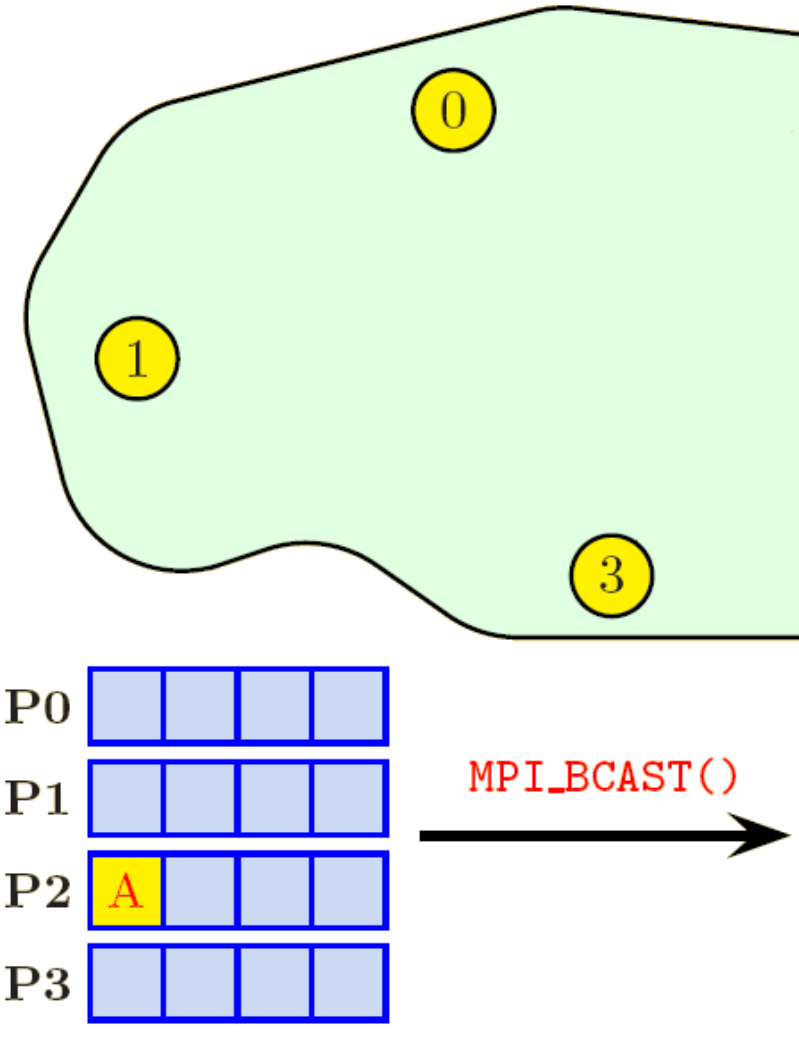
MPI_BARRIER



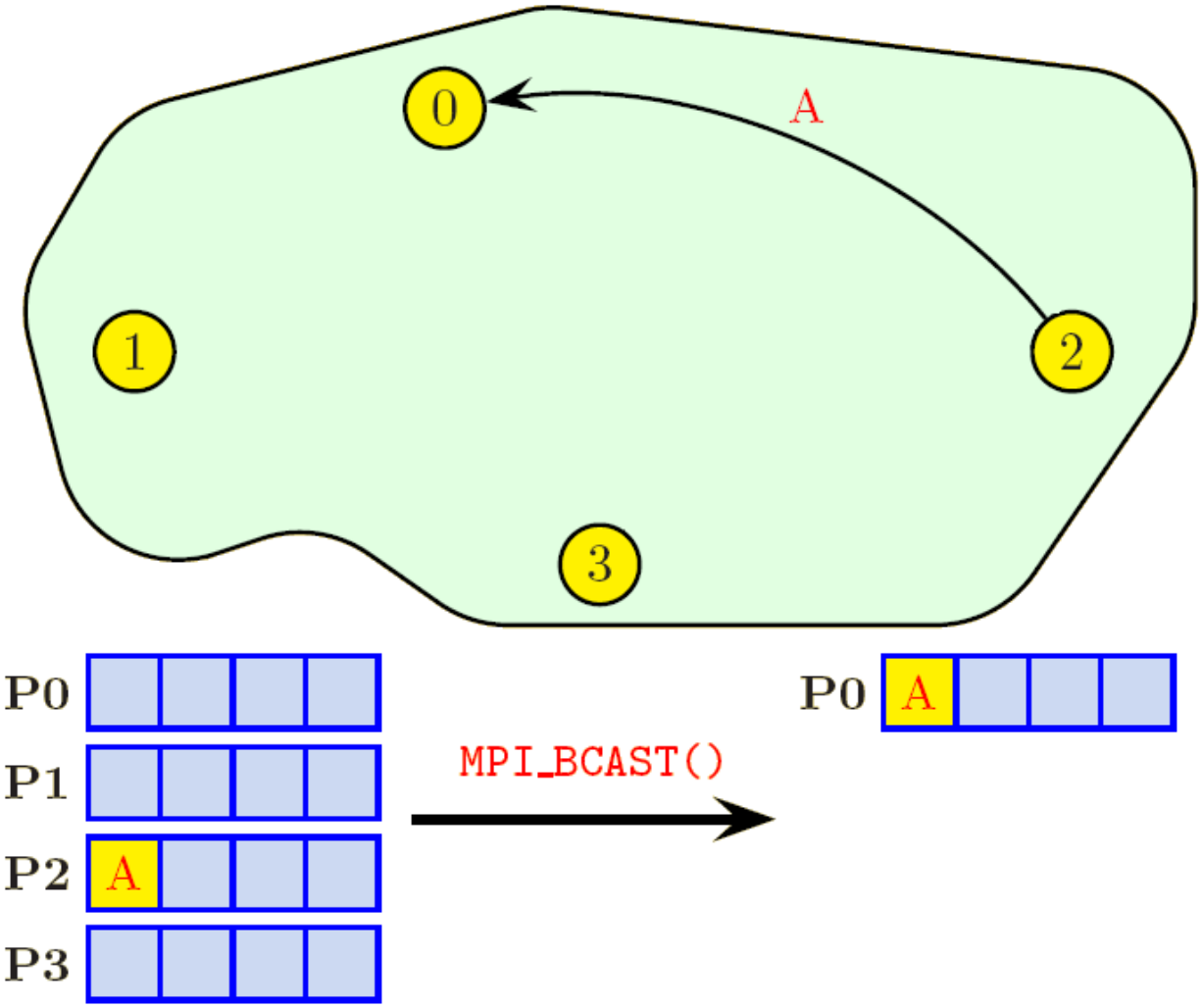
MPI_BARRIER



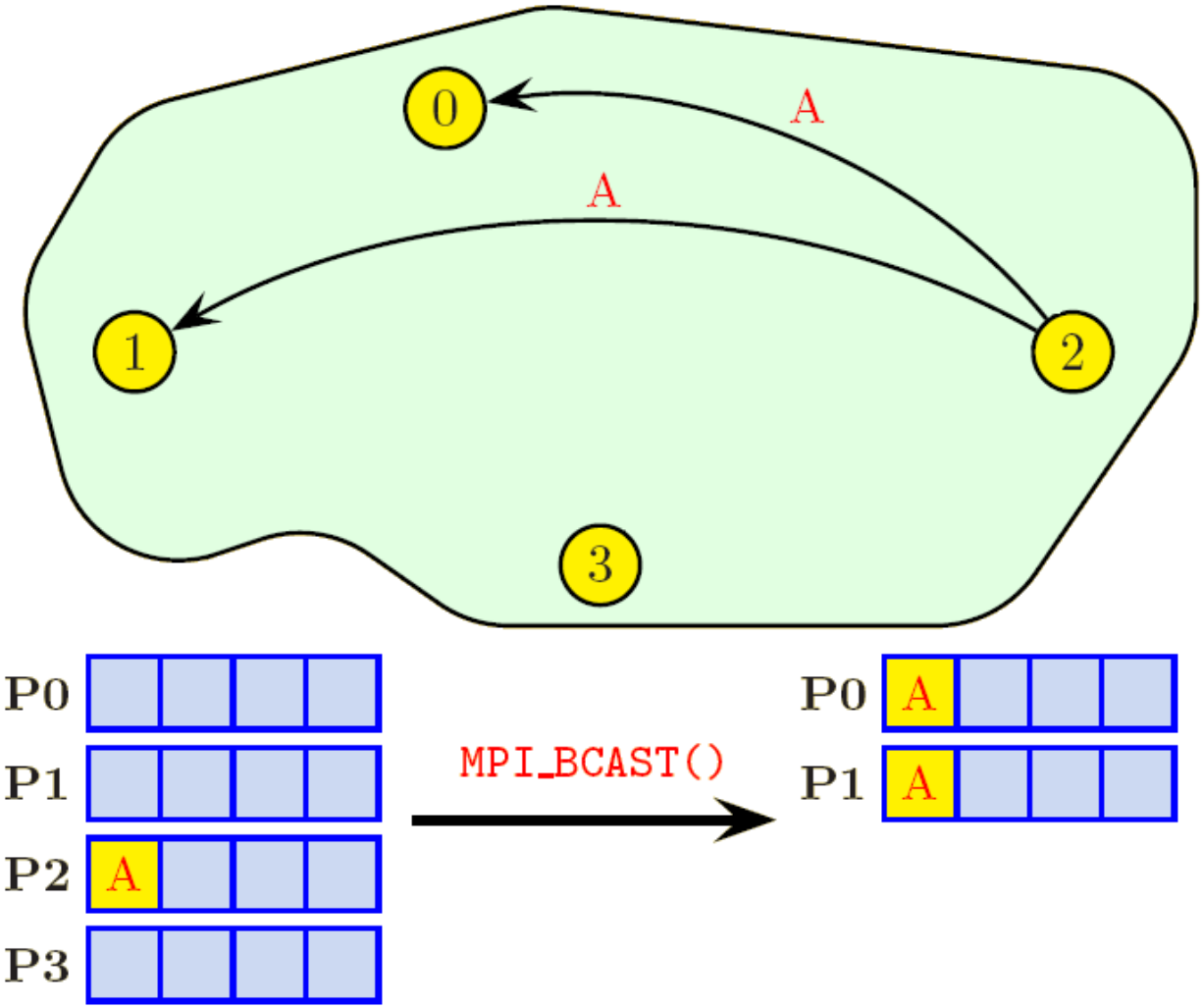
MPI_BCAST



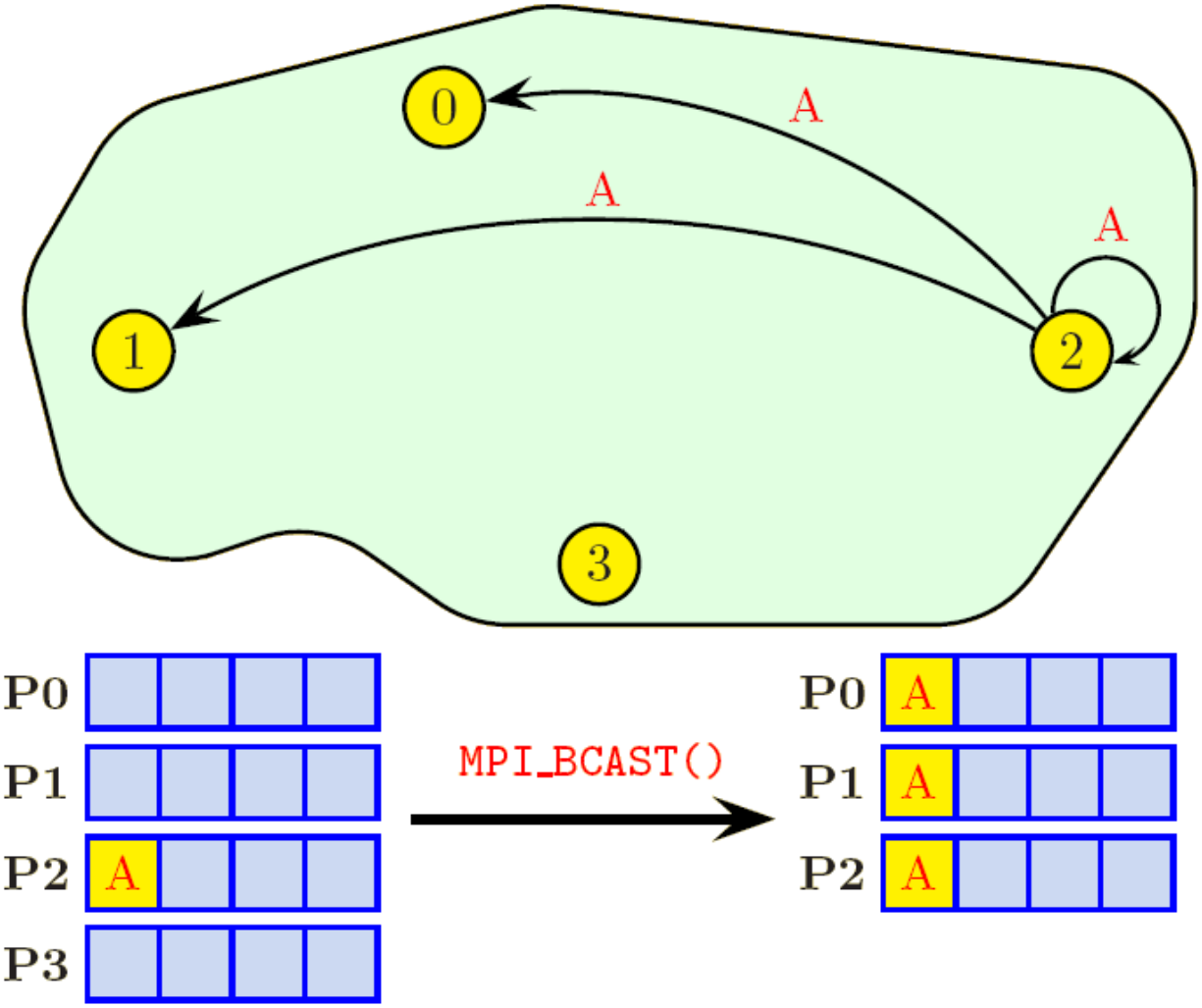
MPI_BCAST



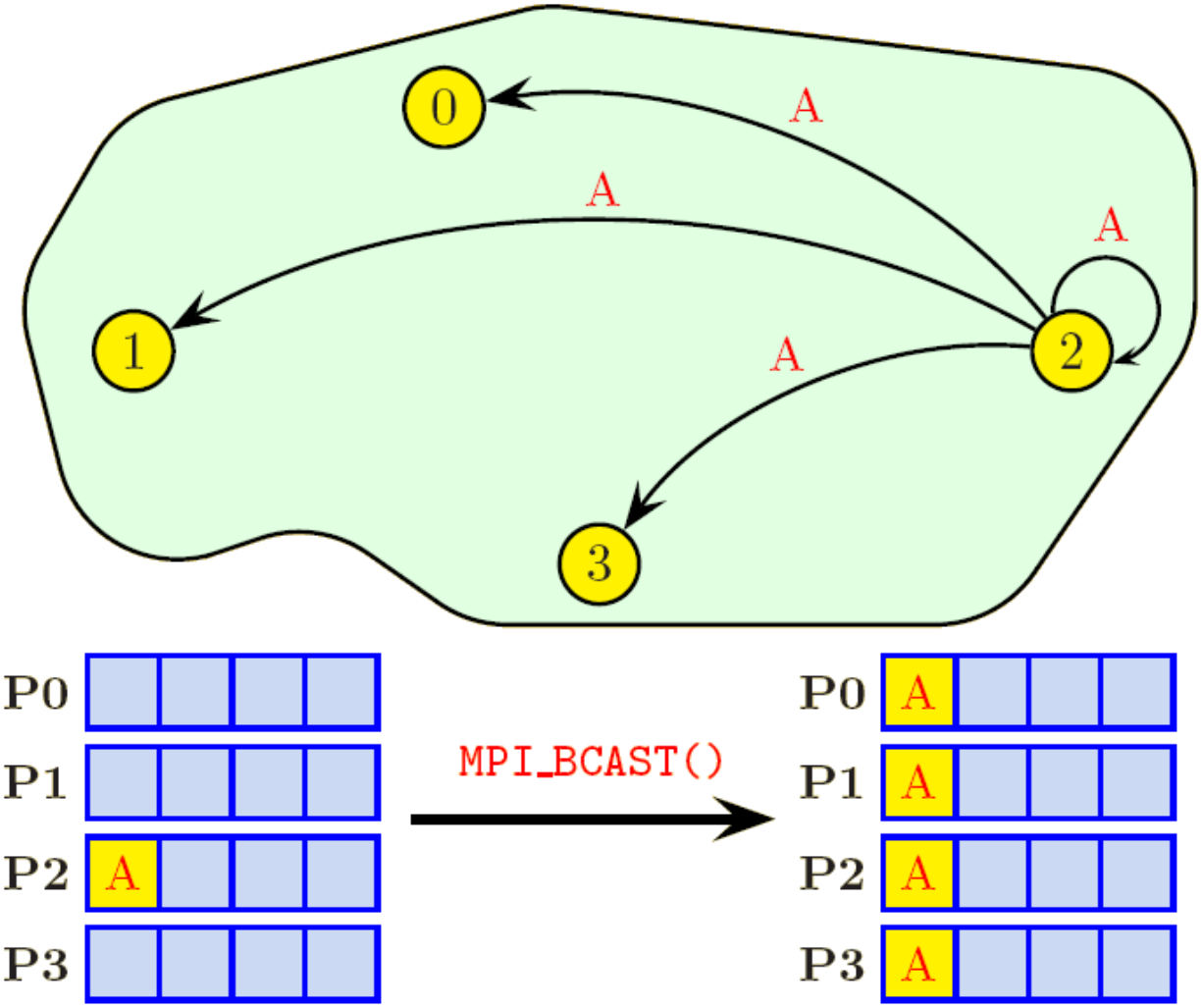
MPI_BCAST



MPI_BCAST



MPI_BCAST



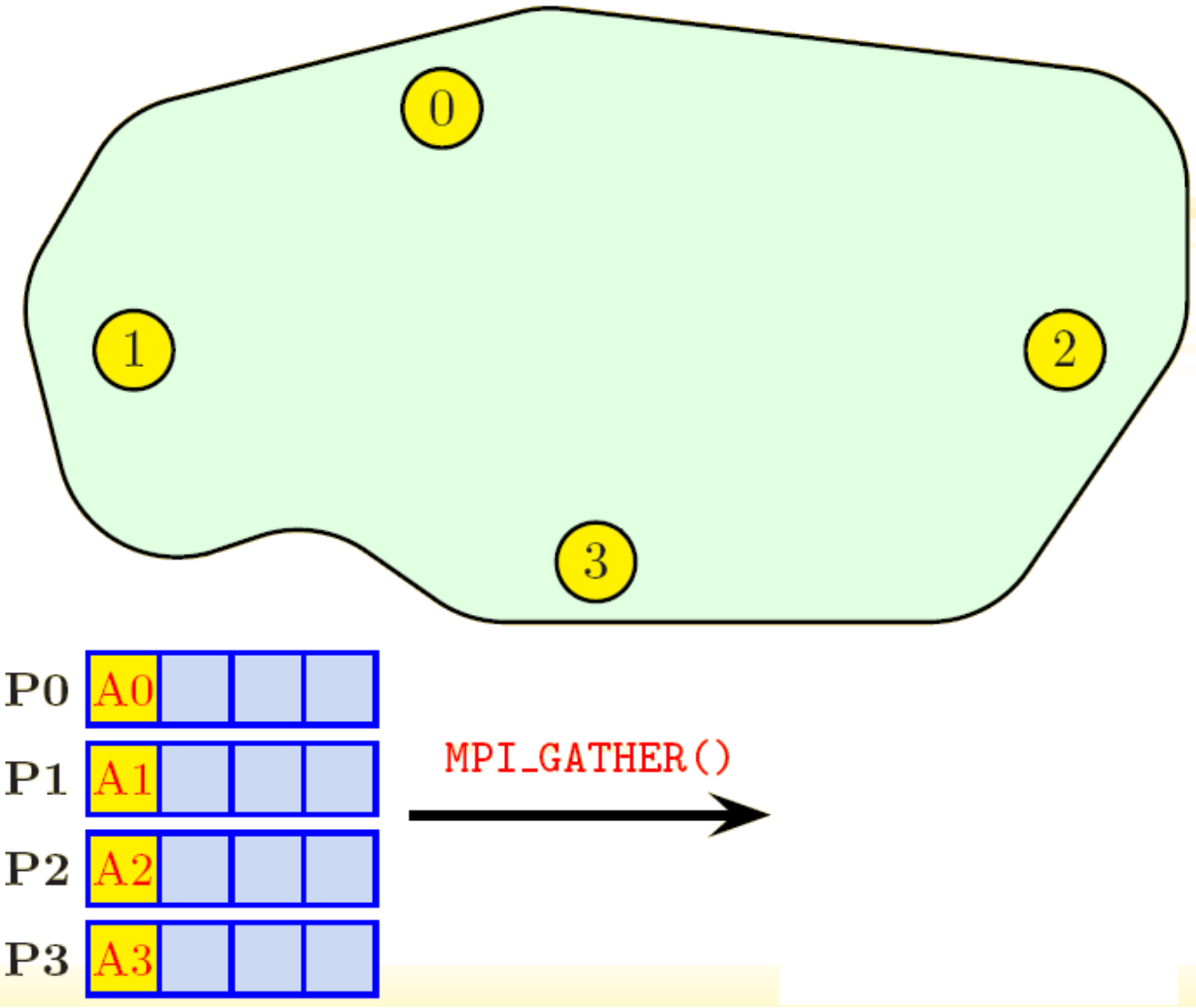
MPI_BCAST

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

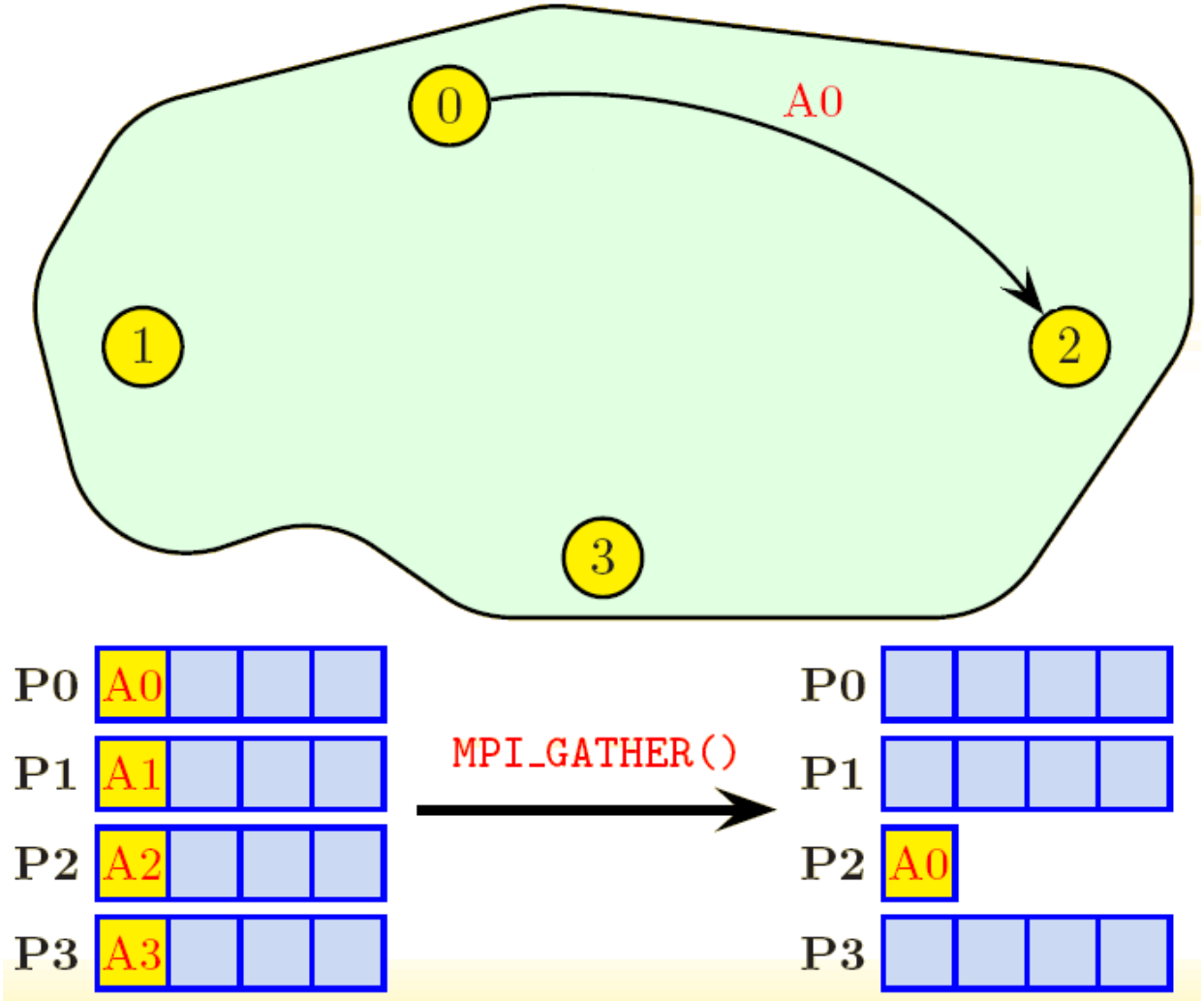
ROOT	**task doing broadcast**	**input**
BUFF	**array being broadcast**	**input/output**
COUNT	**the number of elements**	**input**
MPI_TYPE	**the kind of variable**	**input**

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

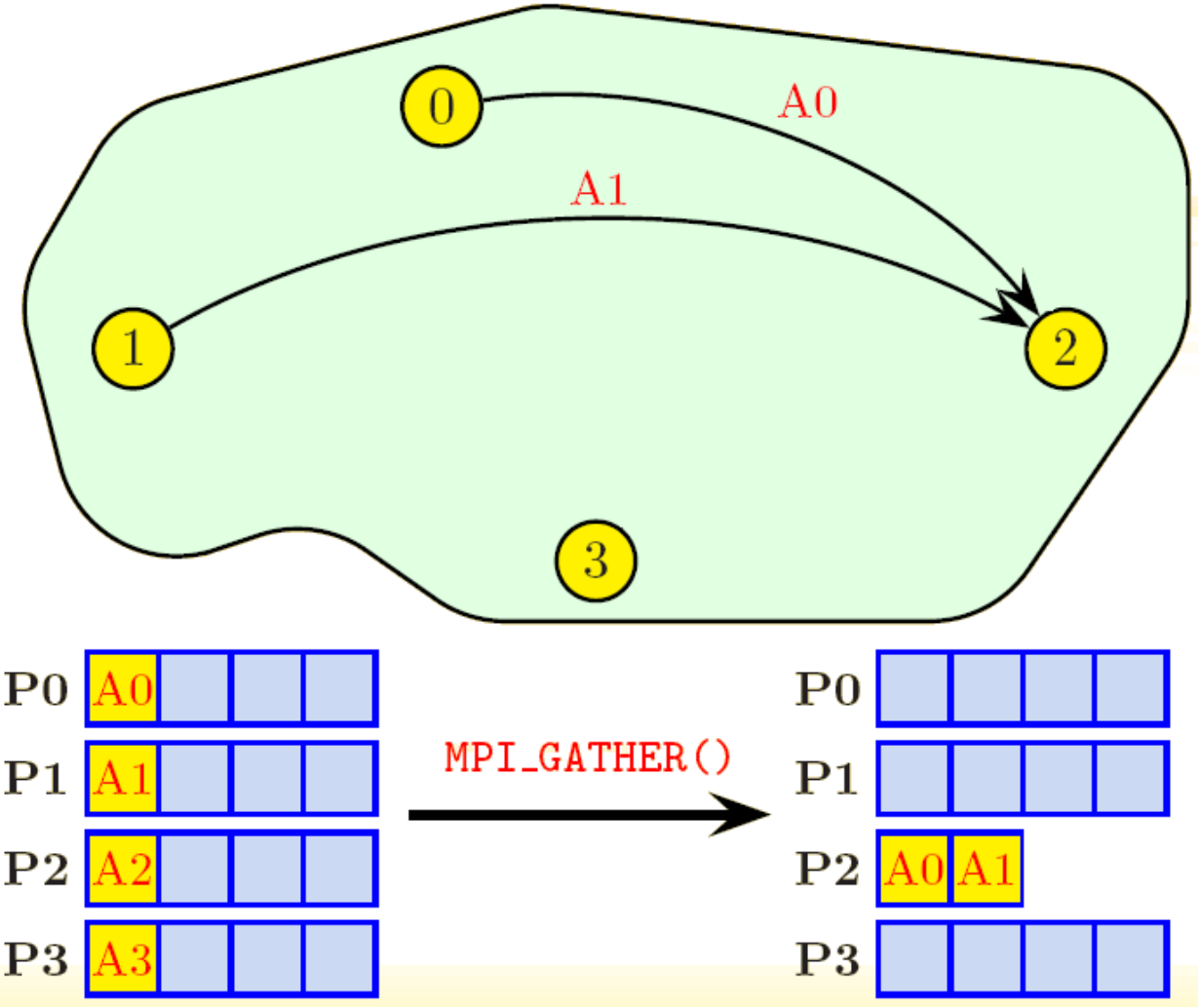
MPI_GATHER



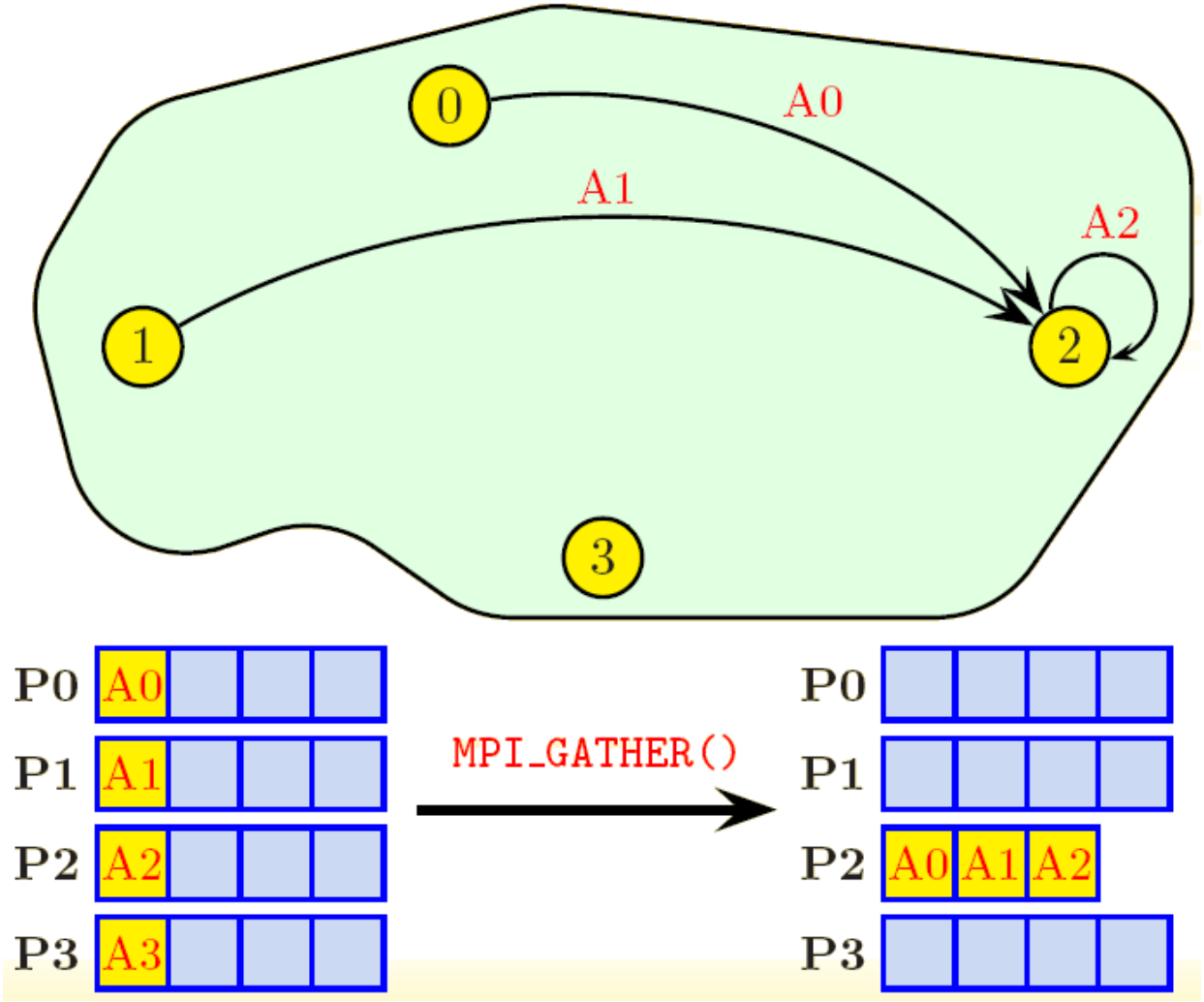
MPI_GATHER



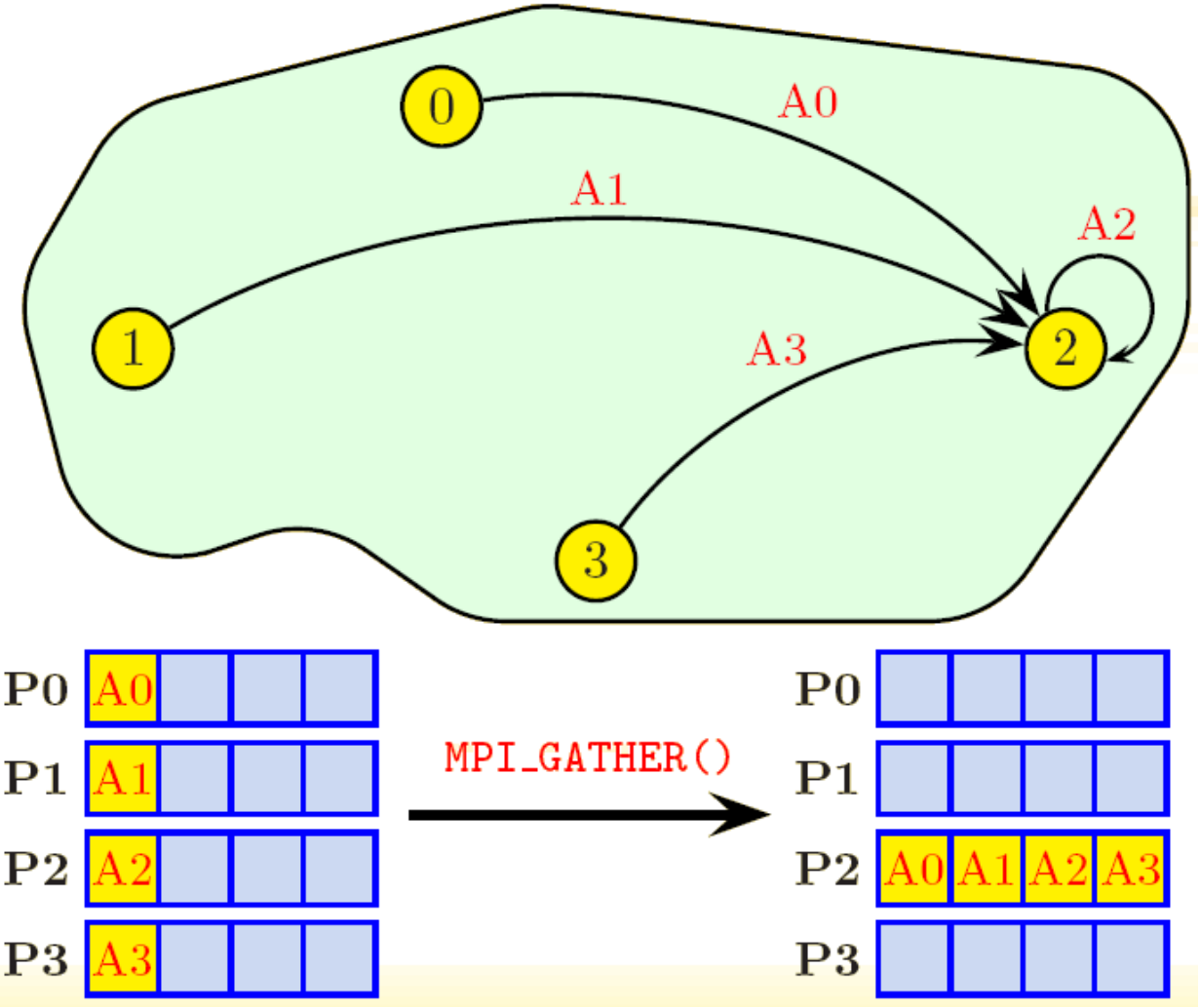
MPI_GATHER



MPI_GATHER



MPI_GATHER


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

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)
```

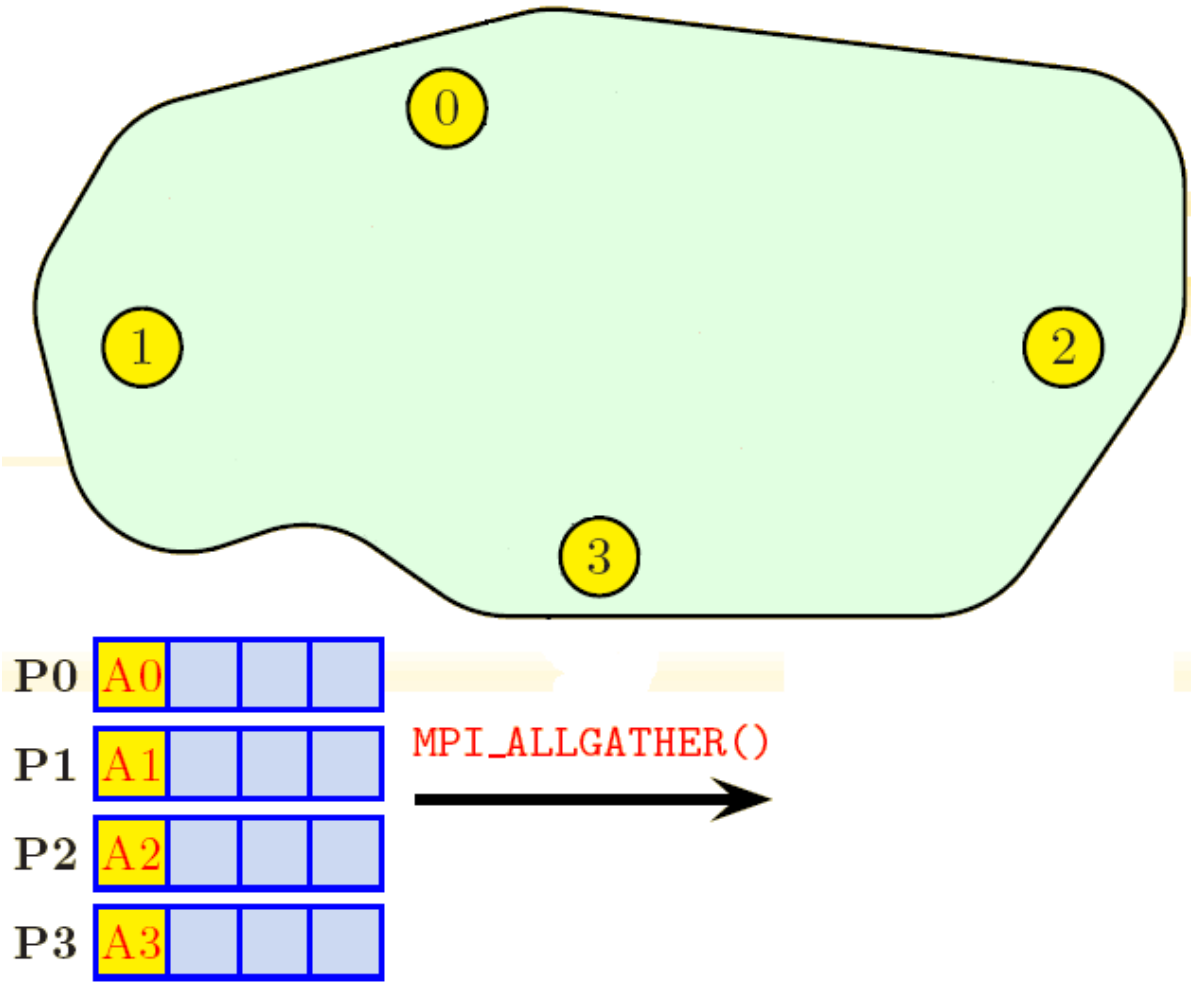
ROOT	**task doing gather**	**input**
SBUFF	**array being sent**	**input**
RBUFF	**array being received**	**output**
[S/R]COUNT	**number of items to/from each task**	**input**

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.

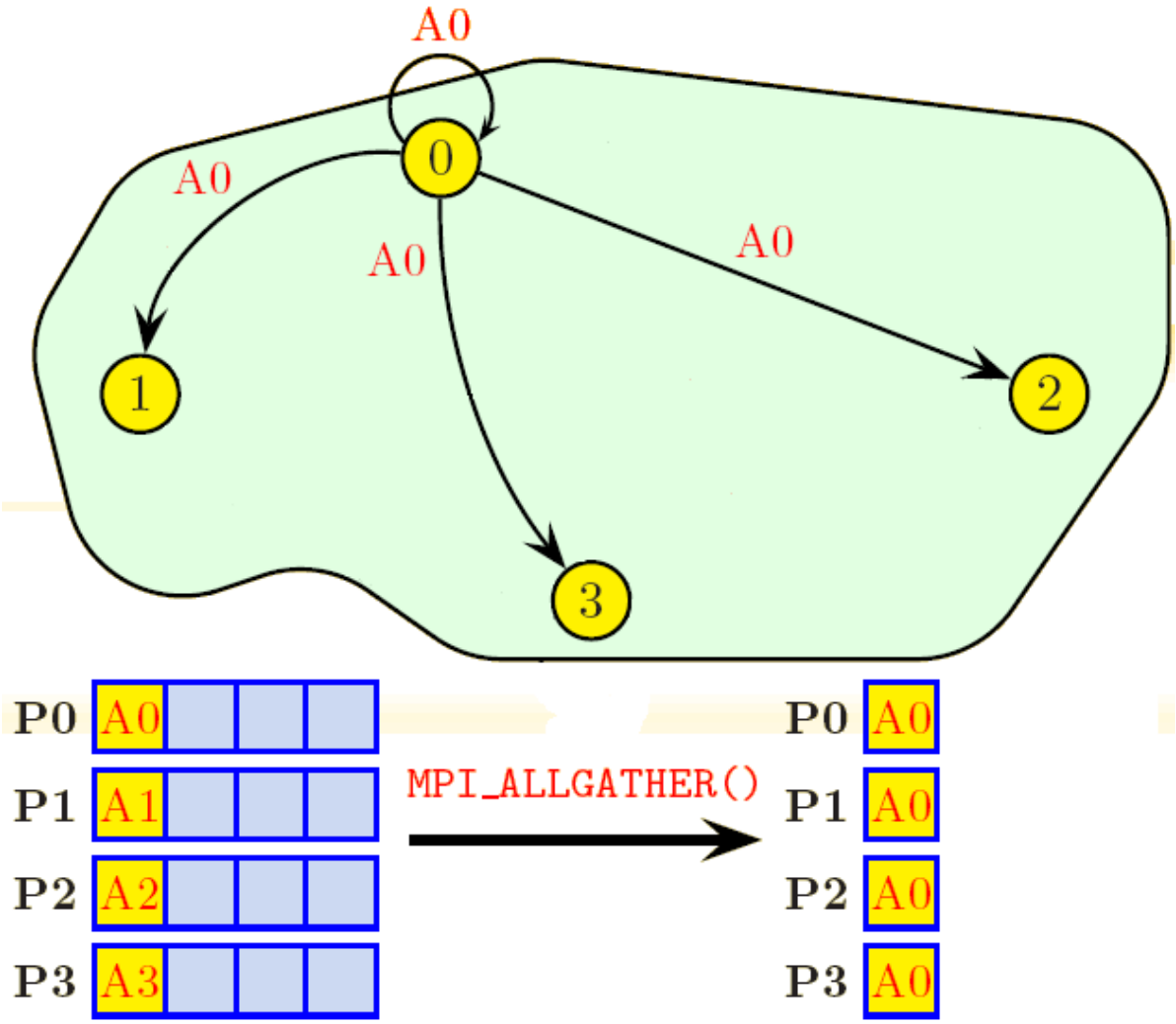
Gather Routines

- **MPI_ALLGATHER**
  - gather arrays of equal length into one array on all 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 all tasks

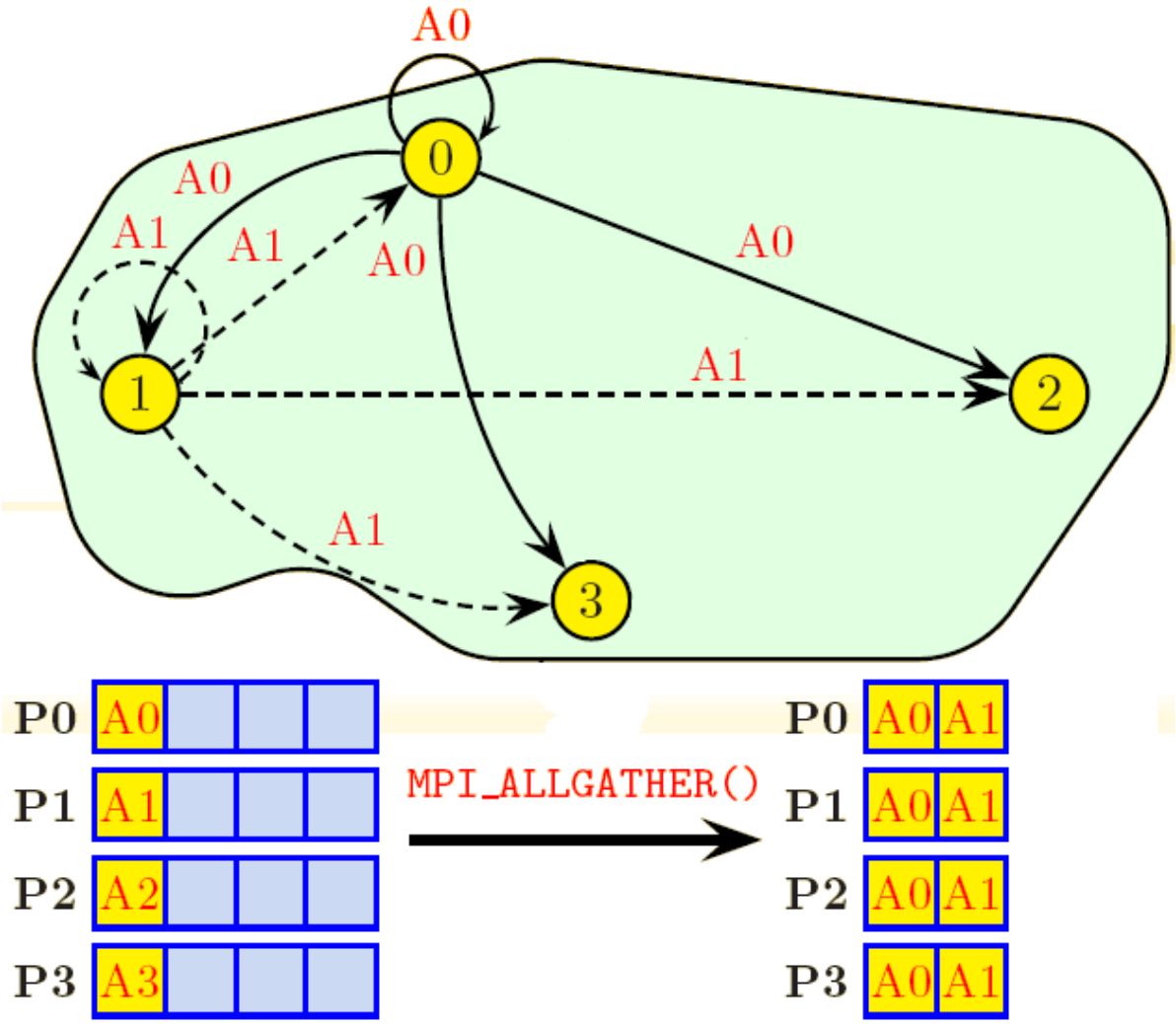
MPI_ALLGATHER



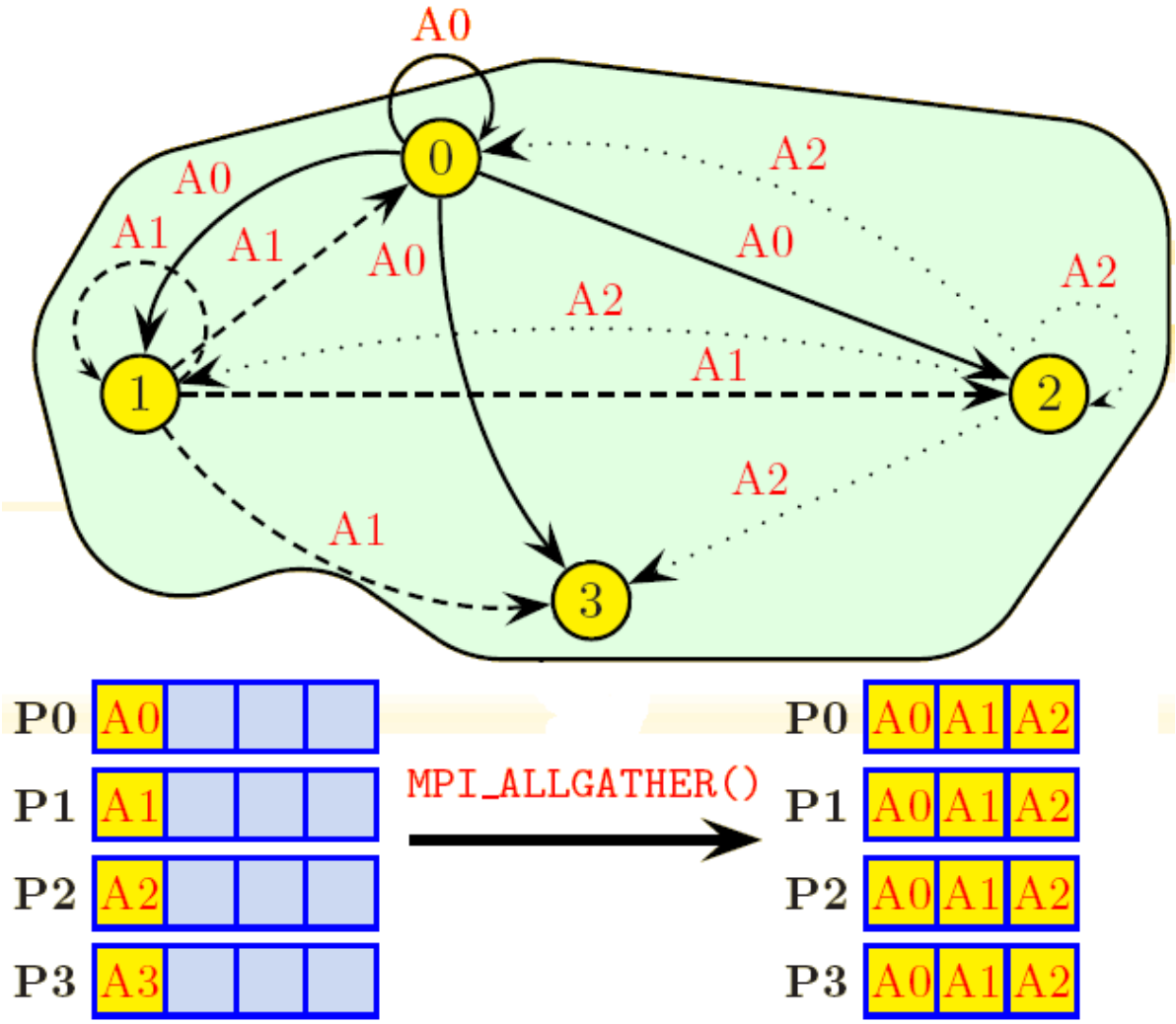
MPI_ALLGATHER



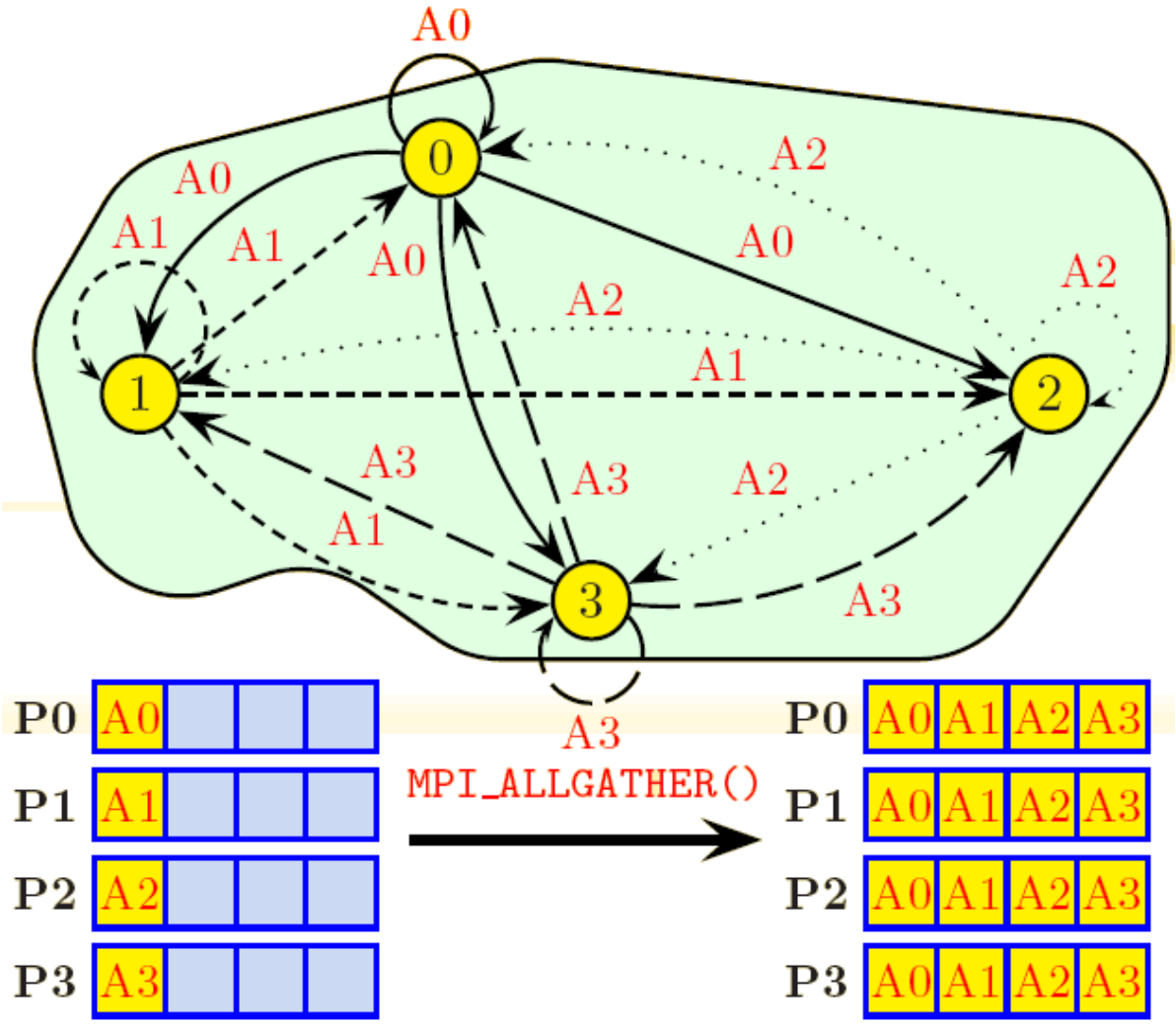
MPI_ALLGATHER



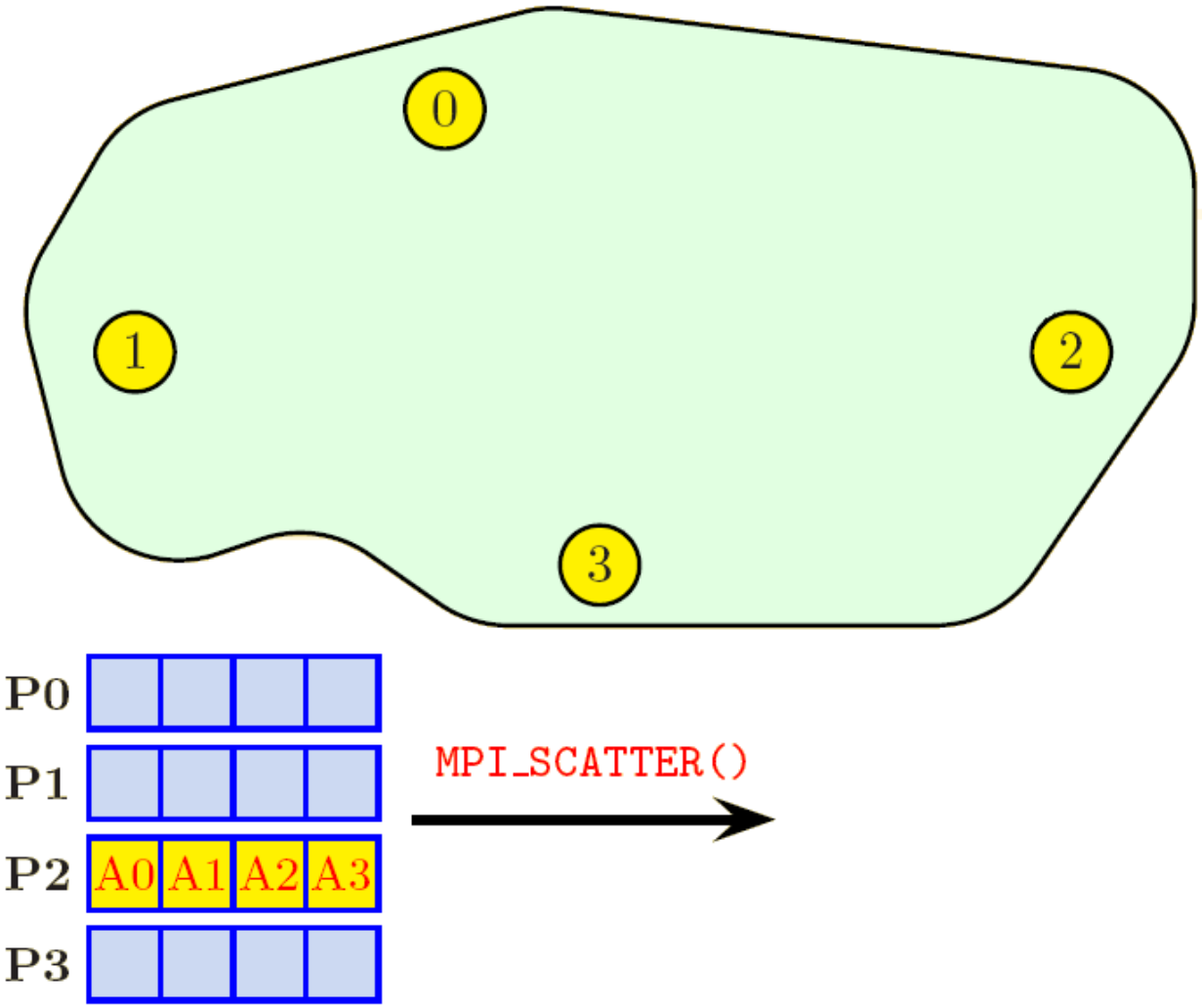
MPI_ALLGATHER



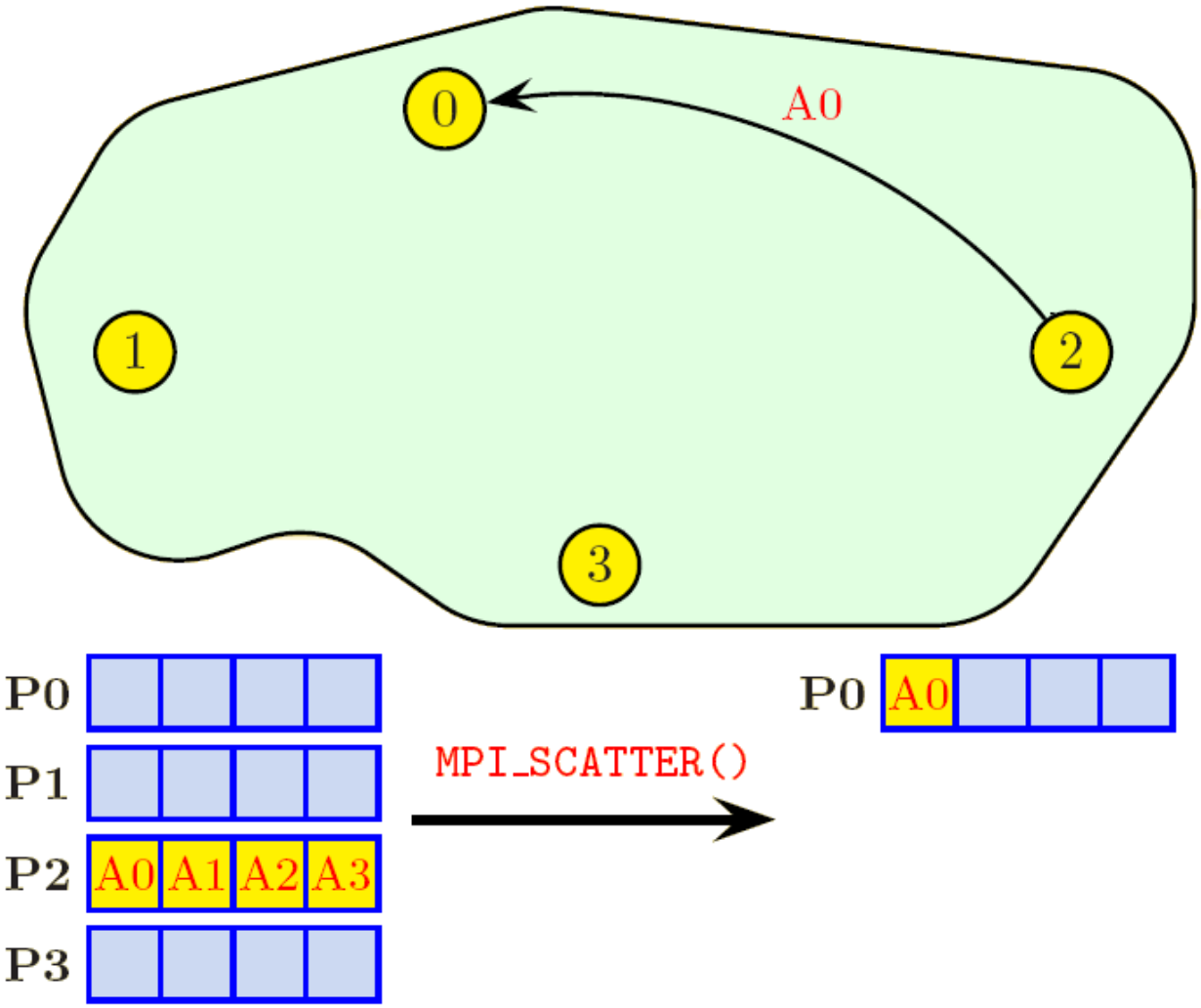
MPI_ALLGATHER



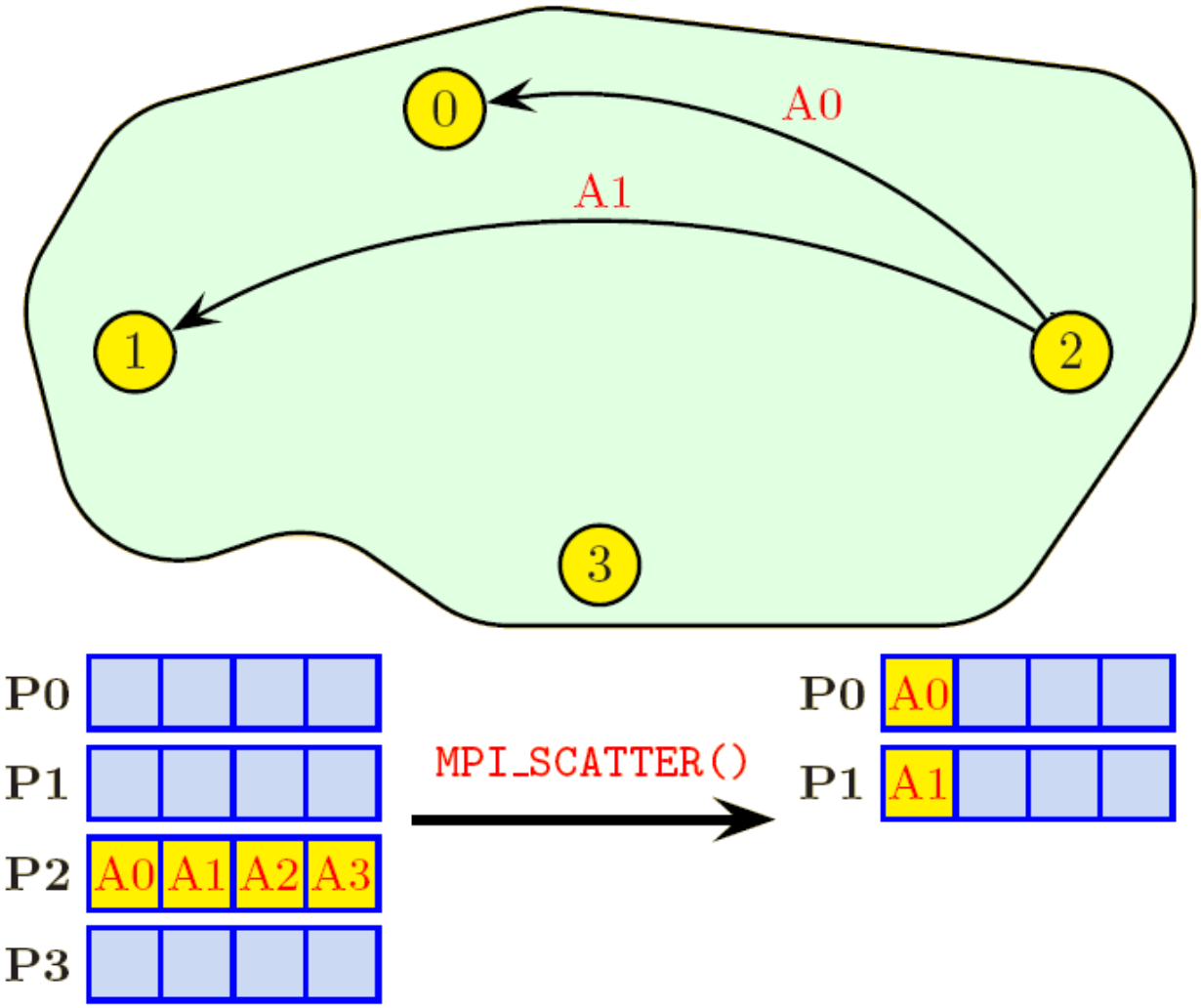
MPI_SCATTER



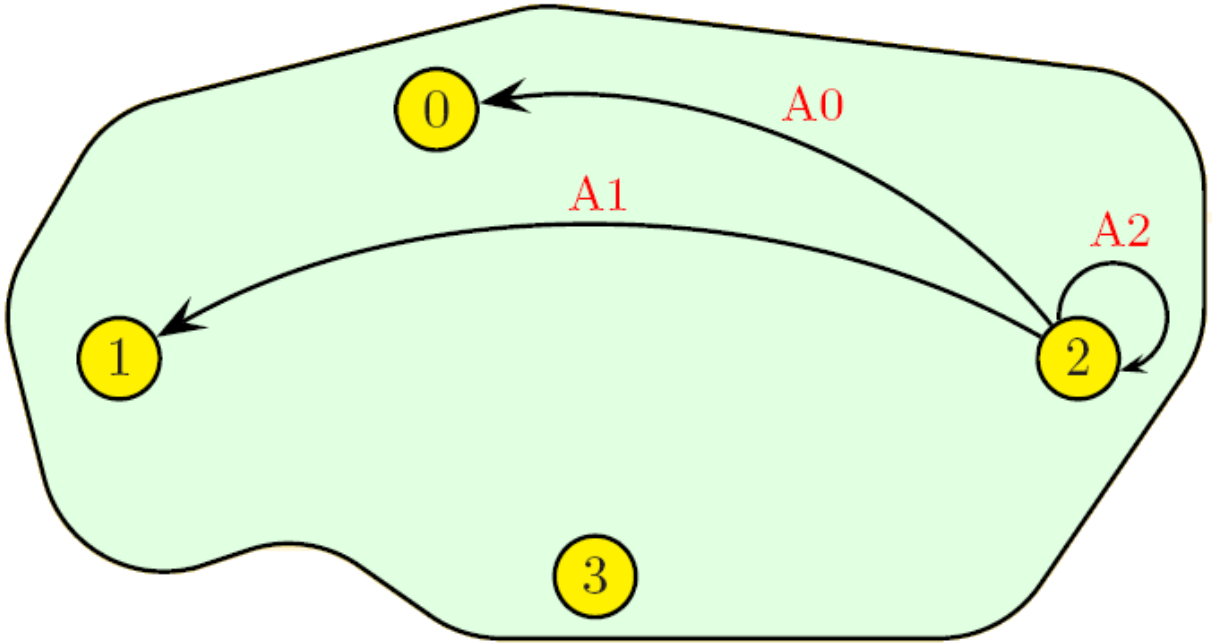
MPI_SCATTER

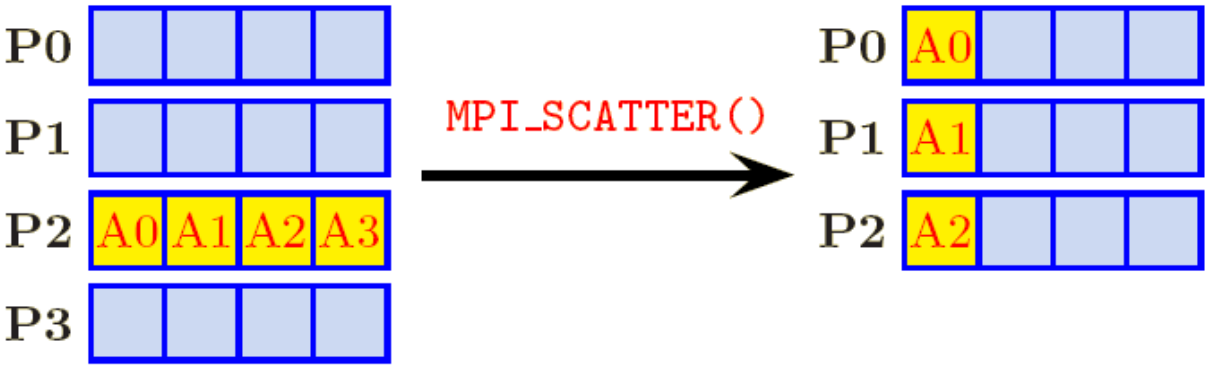


MPI_SCATTER

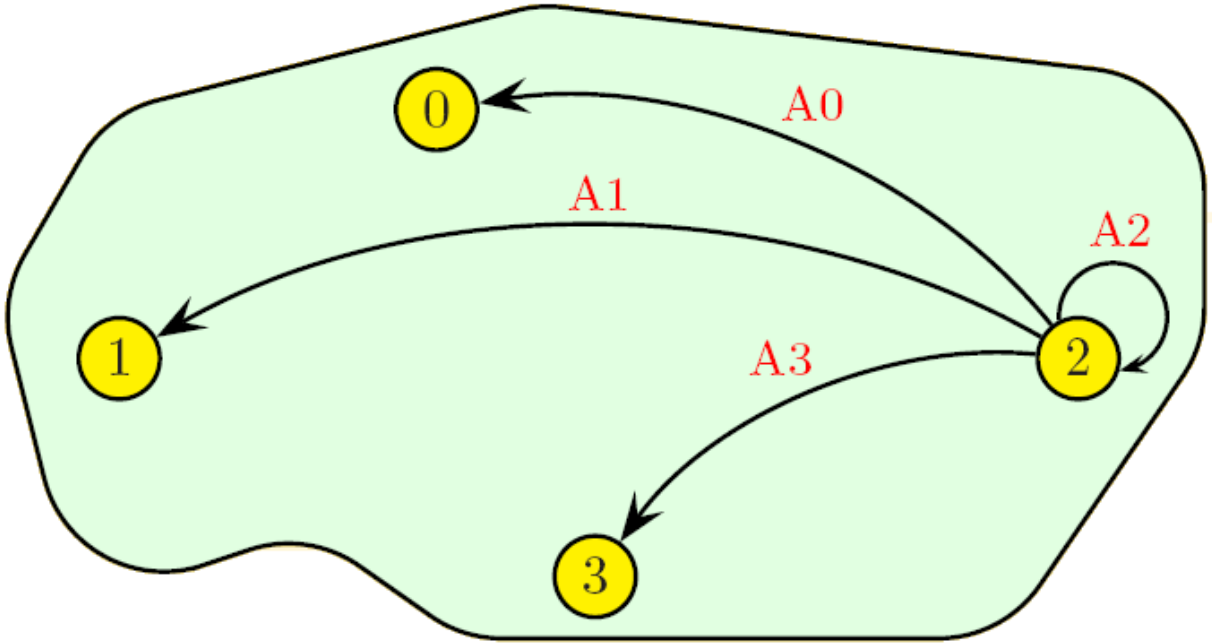


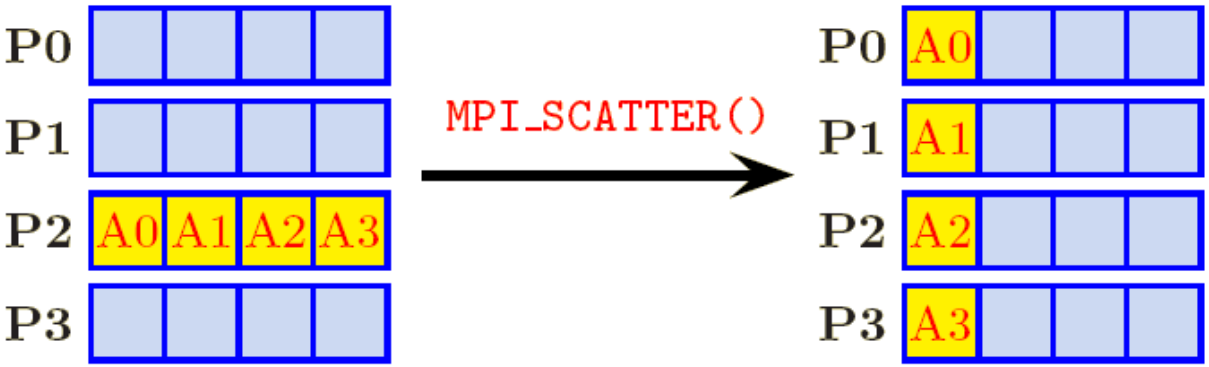
MPI_SCATTER





MPI_SCATTER


0
A0
A1
A2
1
2
A3
3


P0
P1
P2
A0 A1 A2 A3
P3
MPI_SCATTER()
P0 A0
P1 A1
P2 A2
P3 A3

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**

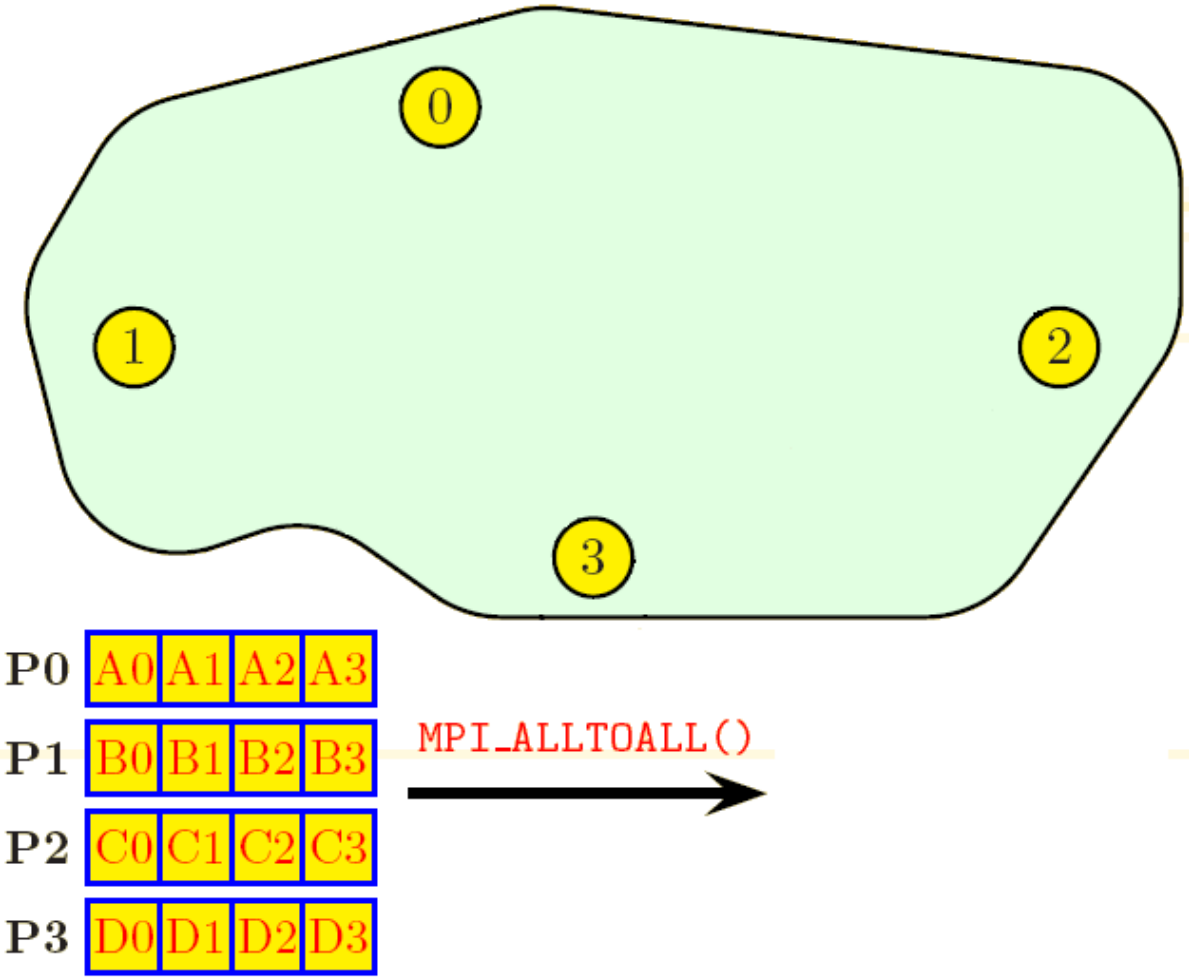
The contents of `sbuff` on task id `root` are equally split and each task receives its part in array `rbuff`. Could also be done by putting `MPI_SEND` in a loop.

Scatter Routines

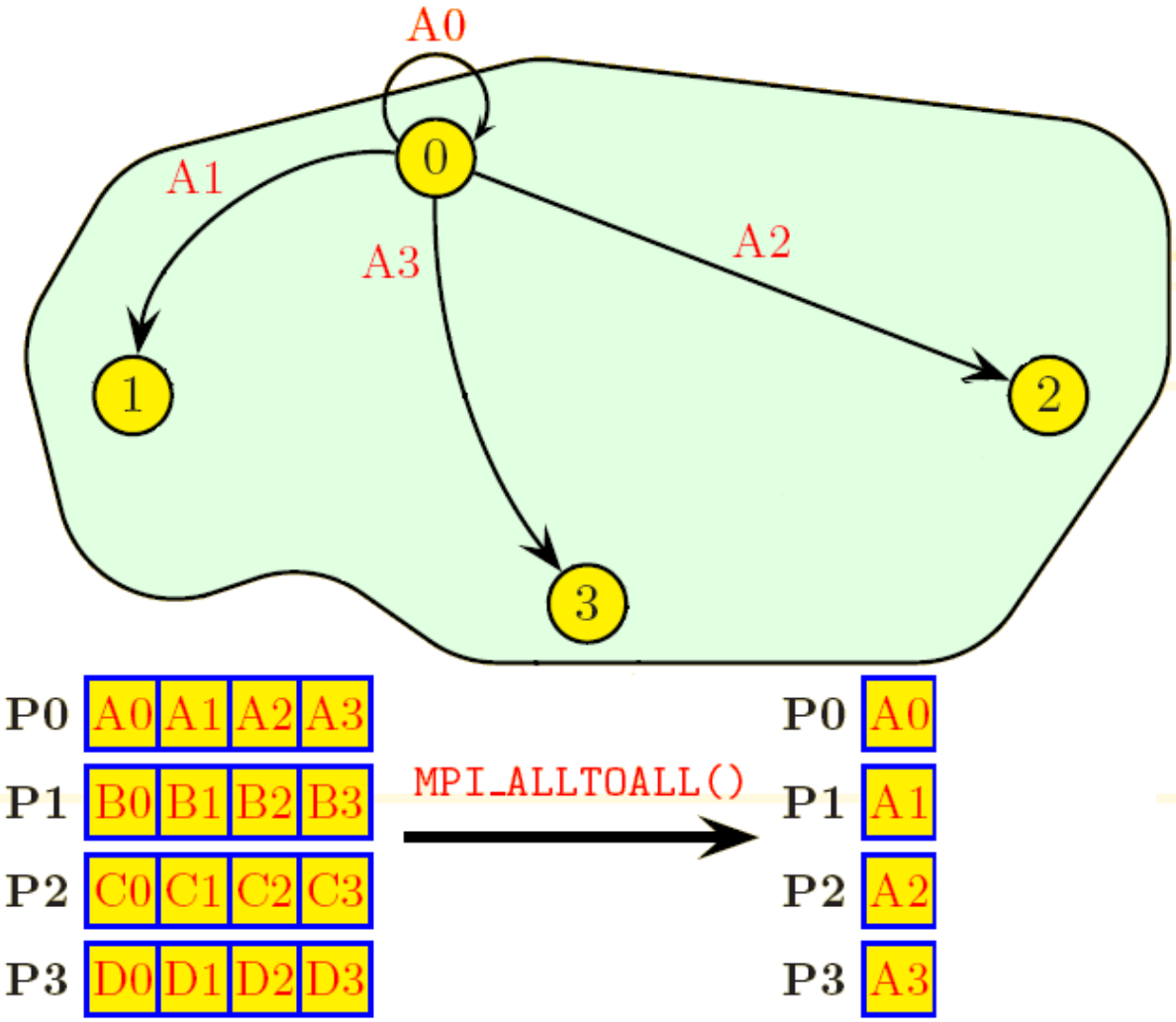
- **`MPI_SCATTER`**
  - divide one array on one task equally amongst all tasks
  - each task receives the same amount of data

- **`MPI_SCATTERV`**
  - divide one array on one task unequally amongst all tasks
  - each task can receive a different amount of data

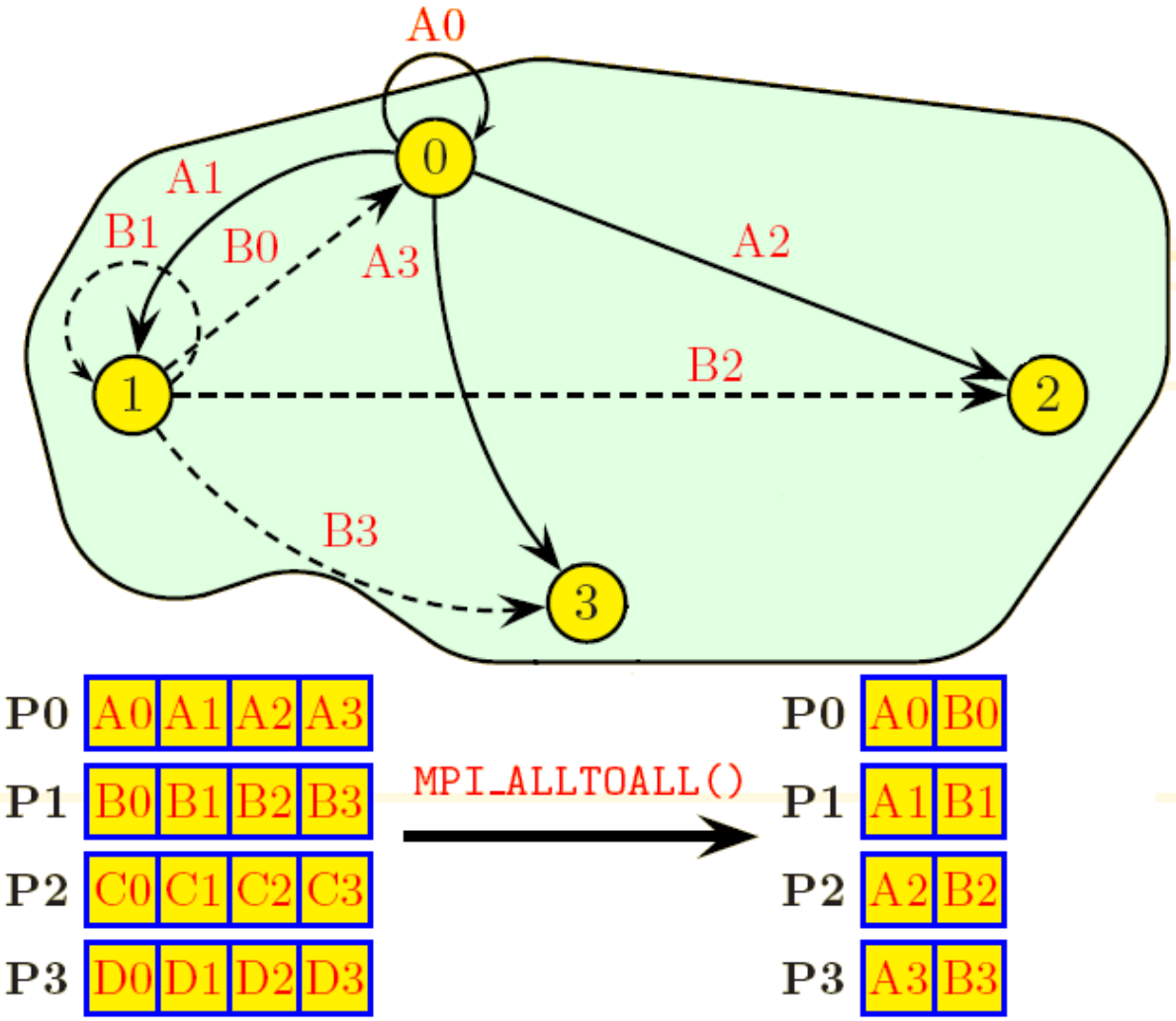
MPI_ALLTOALL



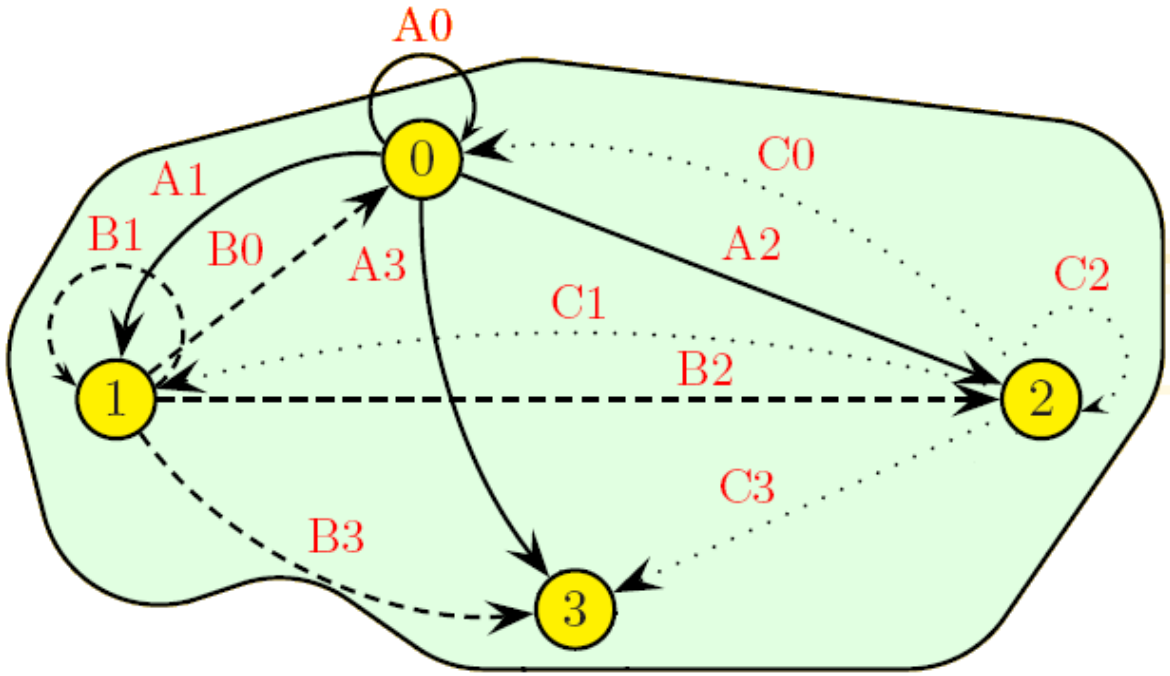
MPI_ALLTOALL

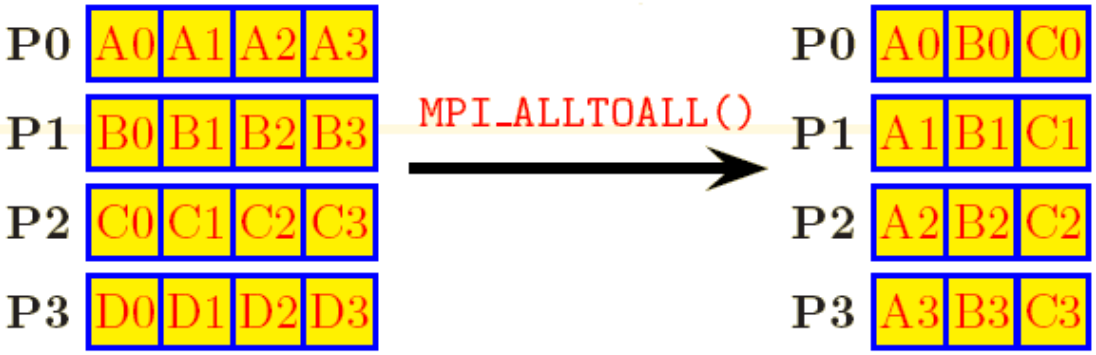


MPI_ALLTOALL

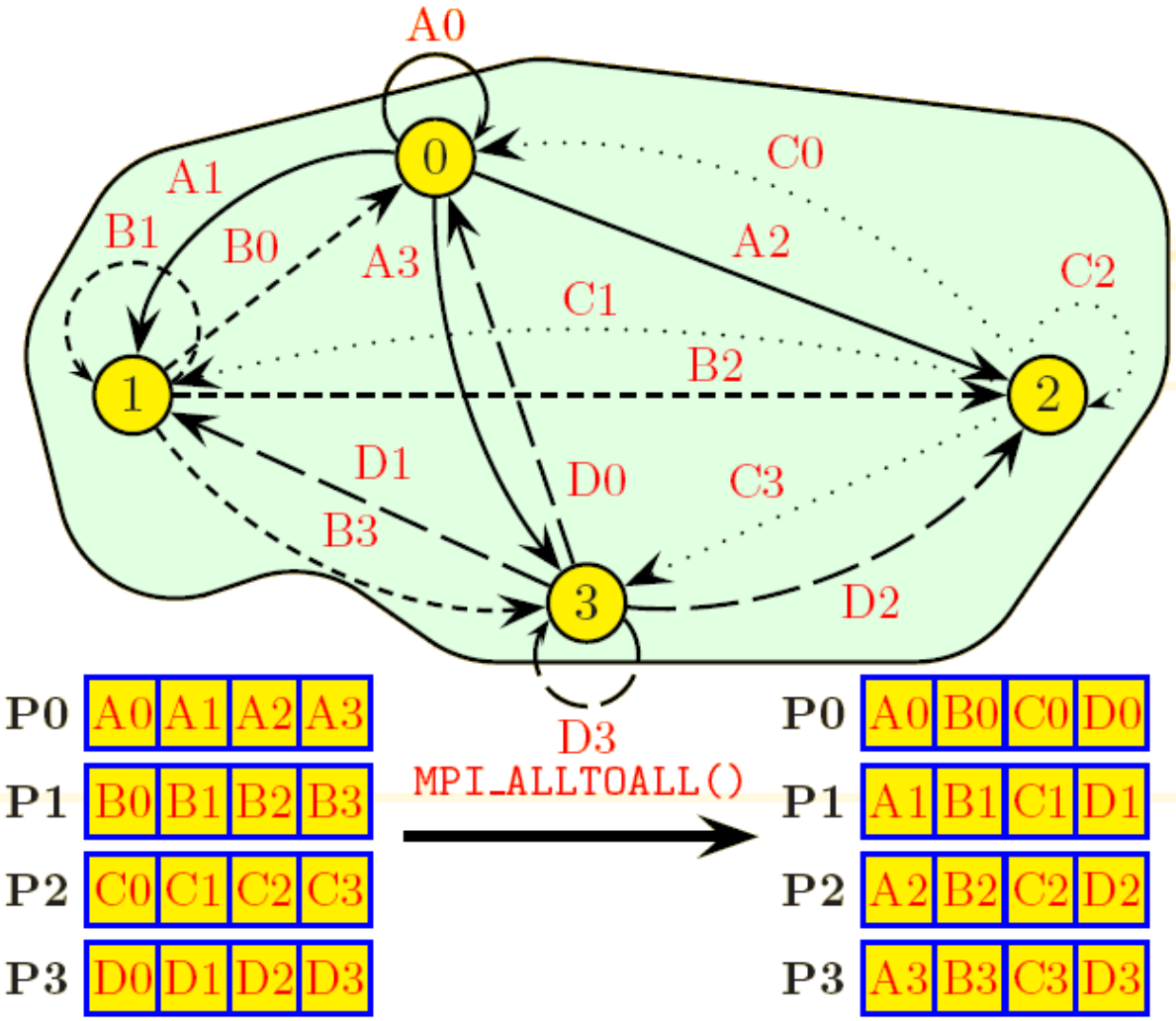


MPI_ALLTOALL





MPI_ALLTOALL



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.

All to All Routines

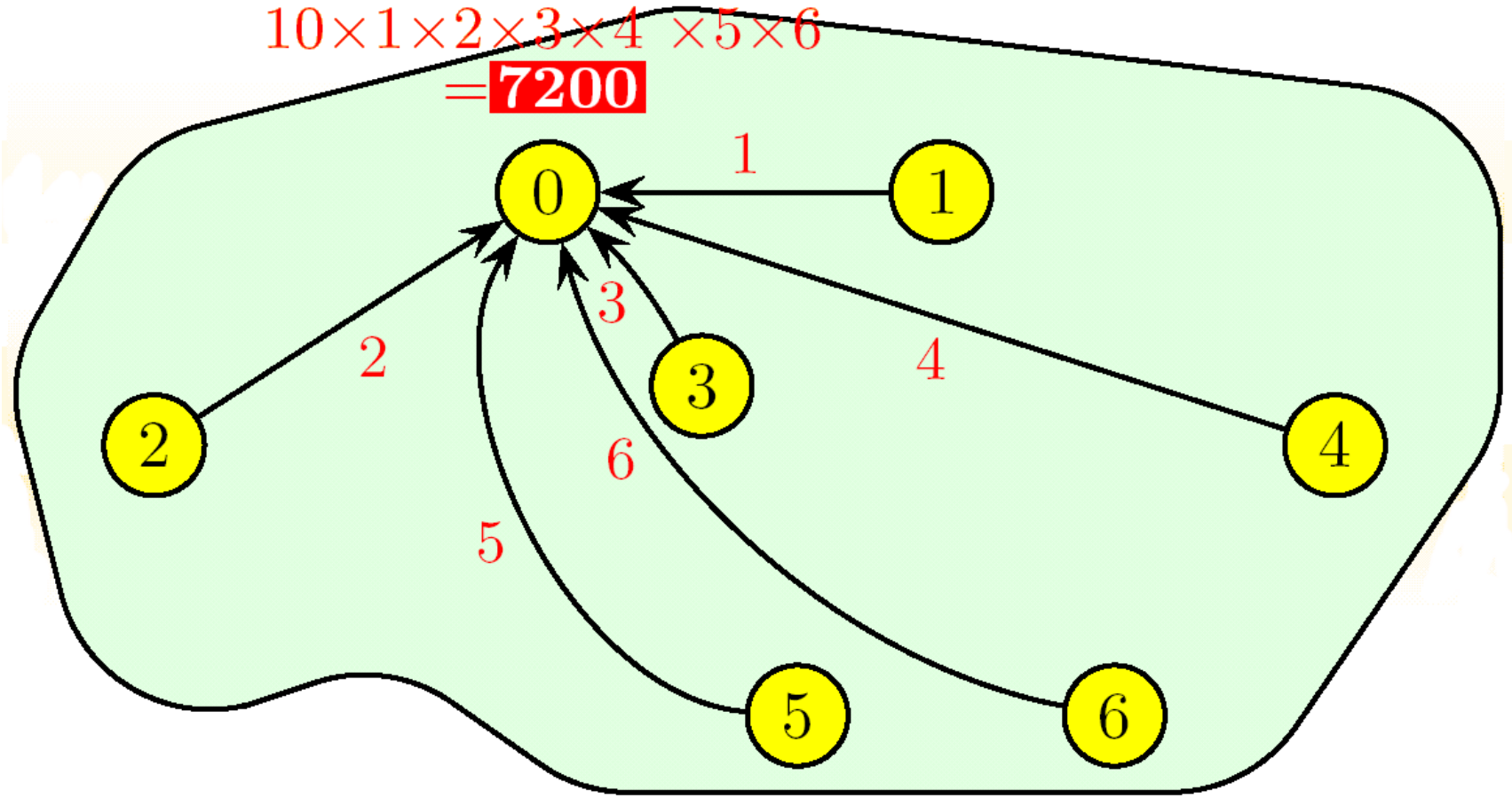
- **MPI_ALLTOALL**
  - every task sends equal length parts of an array to all other tasks
  - every task receives equal parts from all other tasks
  - transpose of data over the tasks

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

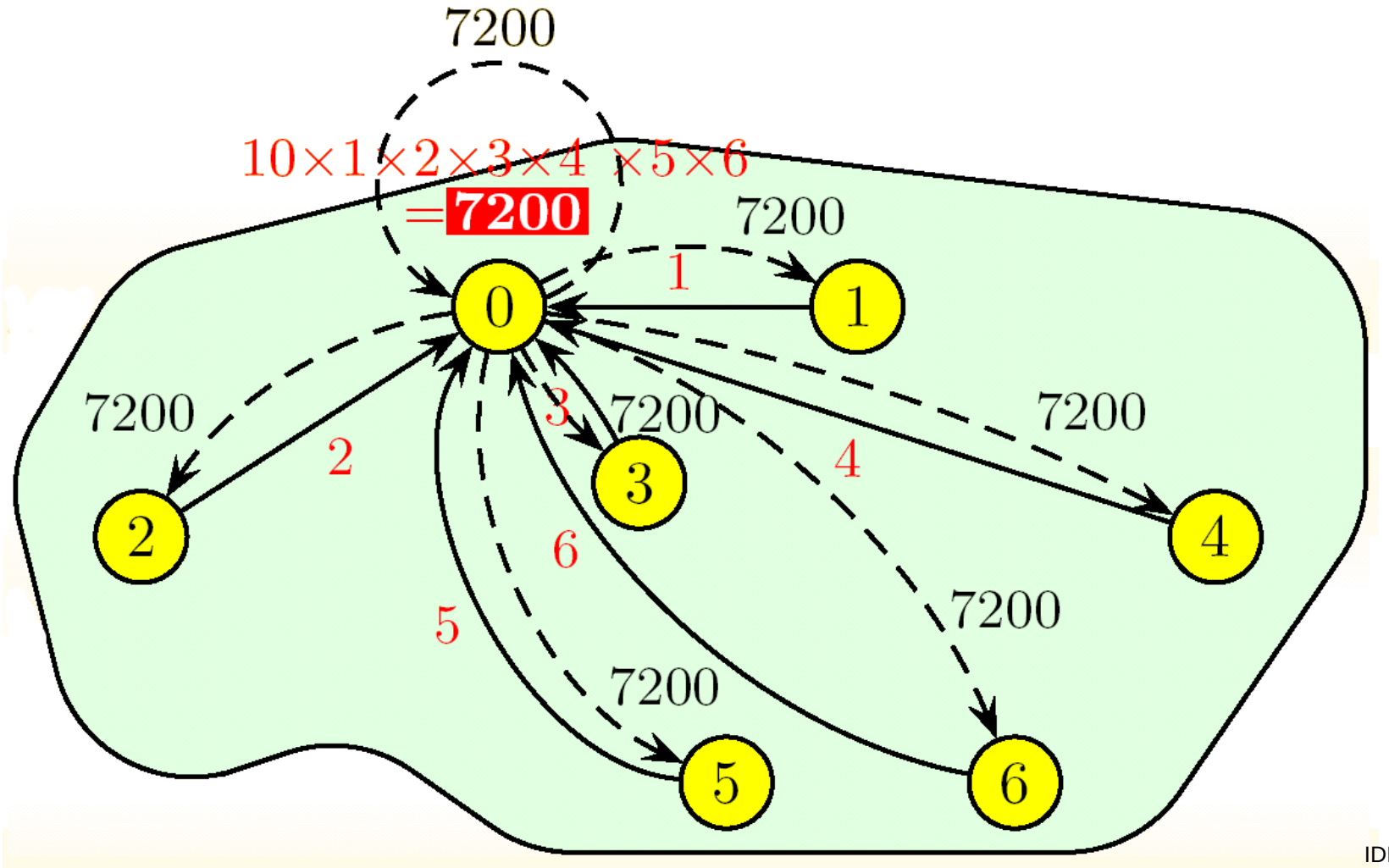
Reduction routines

- 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



MPI_ALLREDUCE



IDRIS-CNRS

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`

Predefined Reduce Operations

MPI_NAME	FUNCTION	MPI_NAME	FUNCTION
MPI_MAX	Maximum	MPI_LOR	Logical OR
MPI_MIN	Minimum	MPI_BOR	Bitwise OR
MPI_SUM	Sum	MPI_LXOR	Logical exclusive OR
MPI_PROD	Product	MPI_BXOR	Bitwise exclusive OR
MPI_LAND	Logical AND	MPI_MAXLOC	Maximum and location
MPI_LOR	Logical OR	MPI_MINLOC	Minimum and Location

Parallelizing your program:

You parallelize your program to run faster.

How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

Time to complete the job on **one** processor

Time to complete the job on **n** processors

Efficiency:

$$E = \frac{S(n)}{n}$$

Tells you how efficiently you parallelize your code

Oversimplified example:

p → fraction of program that can be parallelized

1 - p → fraction of program that cannot be parallelized

n → number of processors

Then the time of running the parallel program will be

1 – p + p/n of the time for running the serial program

80% can be parallelized

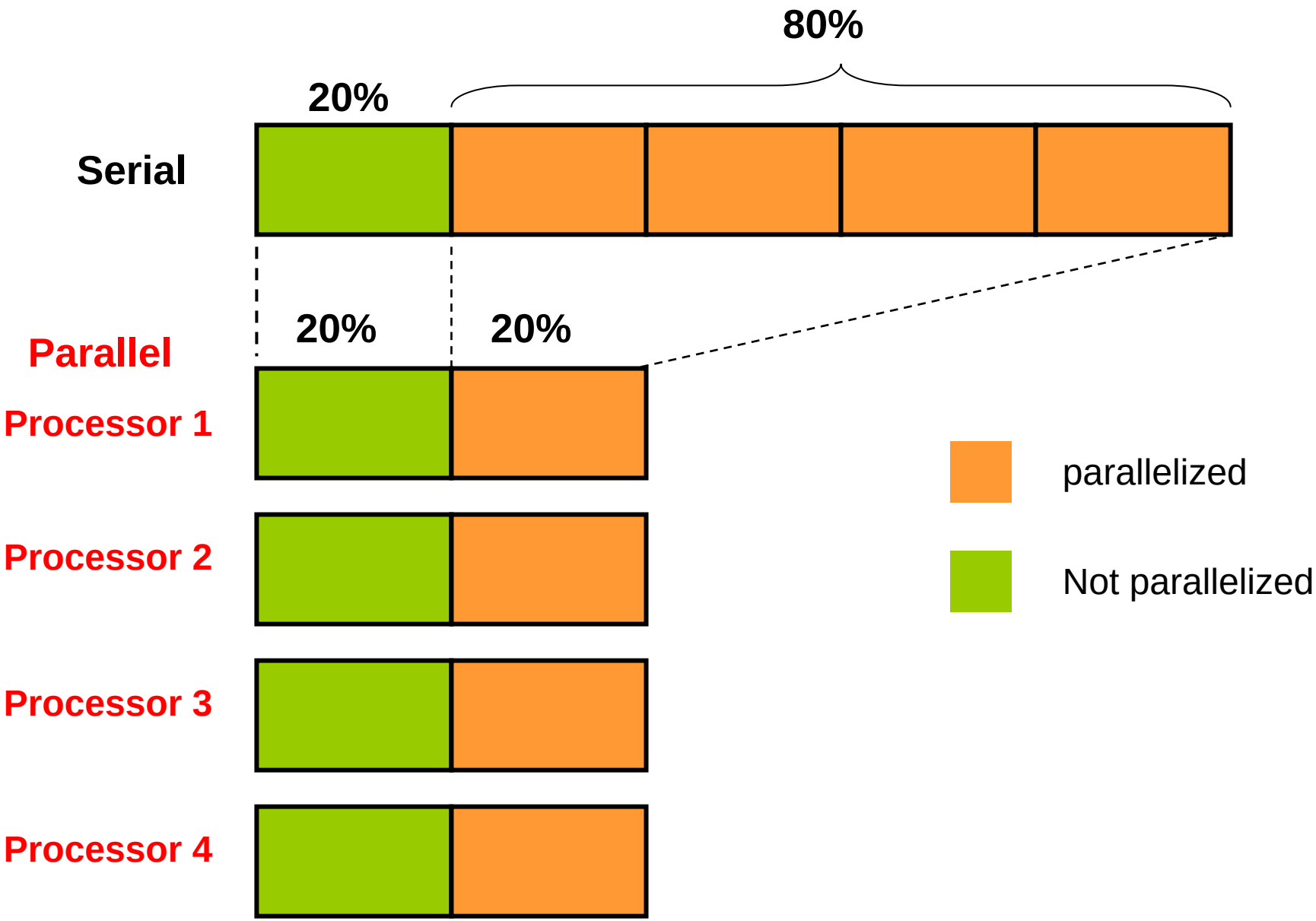
20 % cannot be parallelized

n = 4

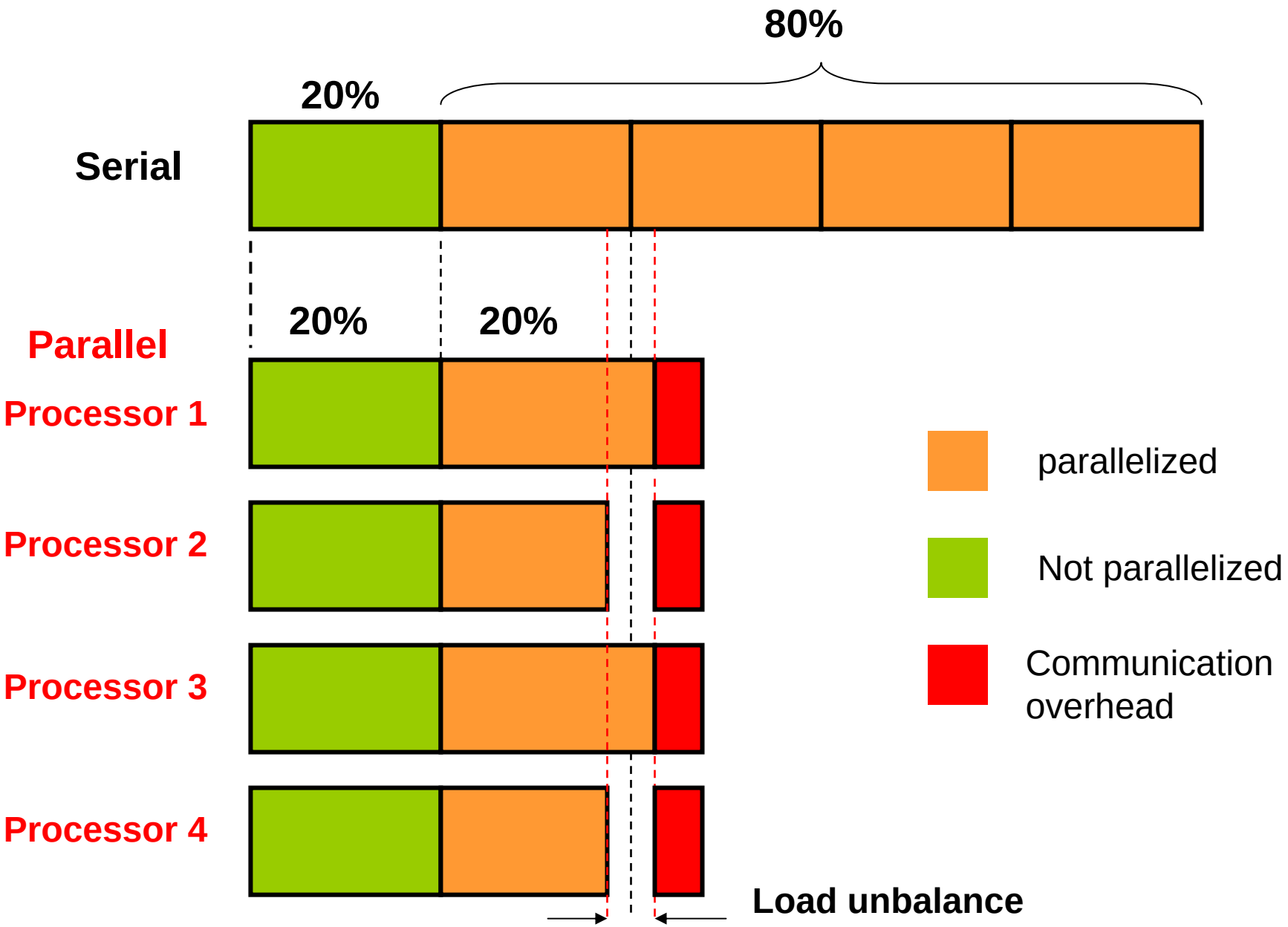
1 - 0.8 + 0.8 / 4 = 0.4 i.e., 40% of the time for running the serial code

You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)

Oversimplified example:



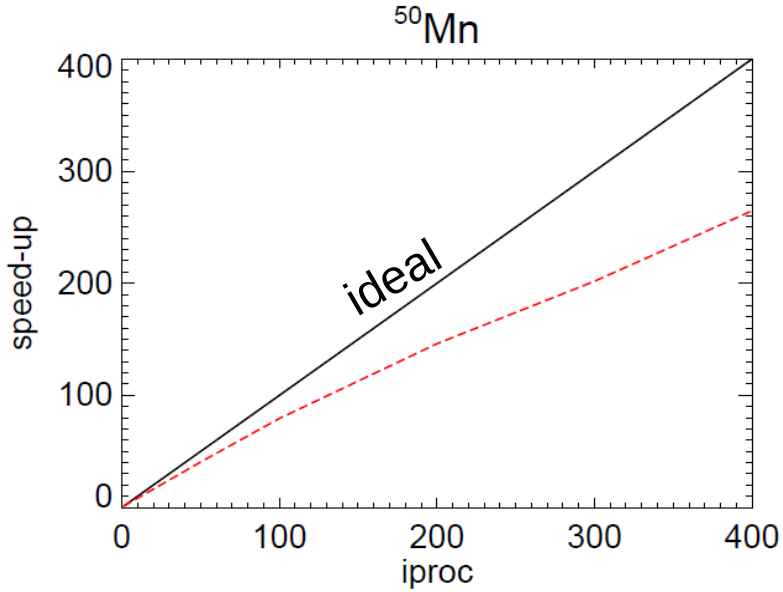
More realistic example:



Realistic example: Speedup of matrix vector multiplication in large scale shell-model calculations

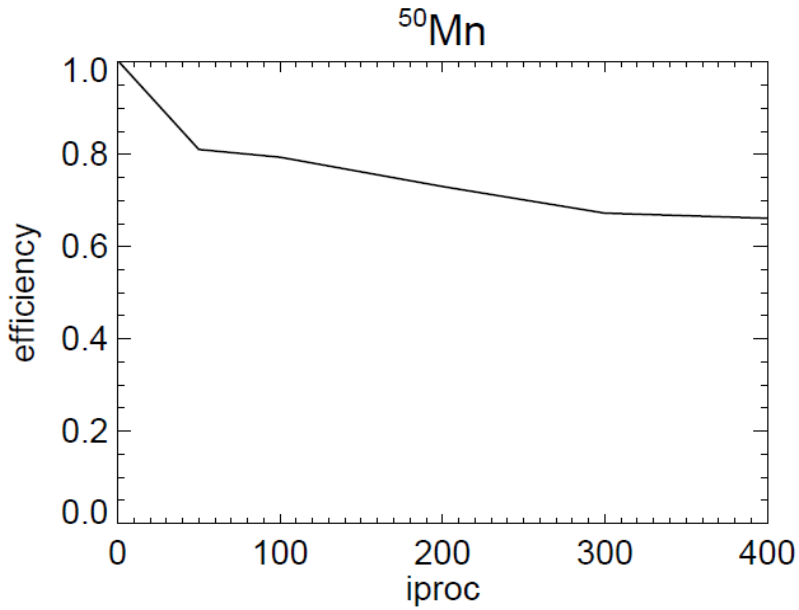
Speed-up

$$S(n) = \frac{T(1)}{T(n)}$$



Efficiency

$$E = \frac{S(n)}{n}$$



Basic guidance for efficient parallelization:

1. Increase the fraction of your program that can be parallelized (identify the most time consuming parts of your program and parallelize them)

2. Balance the parallel workload

3. Minimize the time spent for communication

Parallelizing DO loops:

In almost all of the scientific and technical programs, the hot spots are likely to be found in DO loops. Thus parallelizing DO loops is one of the most important challenges when you parallelize your program. **The basic technique of parallelizing DO loops is to distribute iterations among processors and to let each processor do its portion in parallel.** Usually, the computations within a DO loop involves arrays whose indices are associated with the loop variable. Therefore distributing iterations can often be regarded as dividing arrays and assigning chunks (and computations associated with them) to processors.

Block distribution:

p → number of processors

n → number of iterations

$n = p \times q + r$

r: reminder

q: quotient

Processors 0……r-1 are assigned q+1 iterations

the rest are assigned q iterations

$$n = r (q + 1) + (p – r) q$$

Example:

n = 14, p = 4, q = 3, r = 2

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

The para_range subroutine:

Computes the iteration range for each processor

Usage

```
CALL para_range(n1, n2, nprocs, irank, ista, iend)
```

```
SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
  integer(4) :: n1       ! Lowest value of iteration variable
  integer(4) :: n2       ! Highest value of iteration variable
  integer(4) :: nprocs ! # cores
  integer(4) :: irank    ! Iproc (rank)
  integer(4) :: ista     ! Start of iterations for rank iproc
  integer(4) :: iend     ! End of iterations for rank iproc
  iwork1 = ( n2 - n1 + 1 )  /  nprocs
  iwork2 = MOD(n2 - n1 + 1, nprocs)
  ista = irank * iwork1 + n1 + MIN(irank, iwork2)
  iend = ista + iwork1 - 1
  if ( iwork2 > irank ) iend = iend + 1
End subroutine para_range
```

Simple example:

Add elements of an array a(:)

Serial code:

```
PROGRAM main
Implicit none
Integer(4) :: i, sum
Integer(4), parameter :: n = 1000
Integer(4) :: a(n)
DO i = 1, n
   a(i) = i
ENDDO
sum = 0.0
DO i = 1, n
   sum = sum + a(i)
ENDDO
PRINT *,'sum =',sum
END program main
```

Simple example:

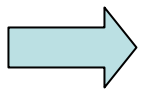
Parallel code:

```
PROGRAM main
INCLUDE ’mpif.h’
PARAMETER (n = 1000)
DIMENSION a(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, ista, iend)
DO i = ista, iend
  a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
  sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum, ssum, 1, MPI_INTEGER,MPI_SUM, 0,MPI_COMM_WORLD, ierr)
sum = ssum
IF (myrank == 0) PRINT *,’sum =’,sum
CALL MPI_FINALIZE(ierr)
END
```

Cyclic Distribution:

In cyclic distribution, the iterations are assigned to processes in a round-robin fashion.

```
DO i = n1, n2
   computation
ENDDO
```



```
DO i = n1+myrank, n2,nproc
   computation
ENDDO
```

Example: Distributing 14 iterations over 4 cores in round-robin fashion

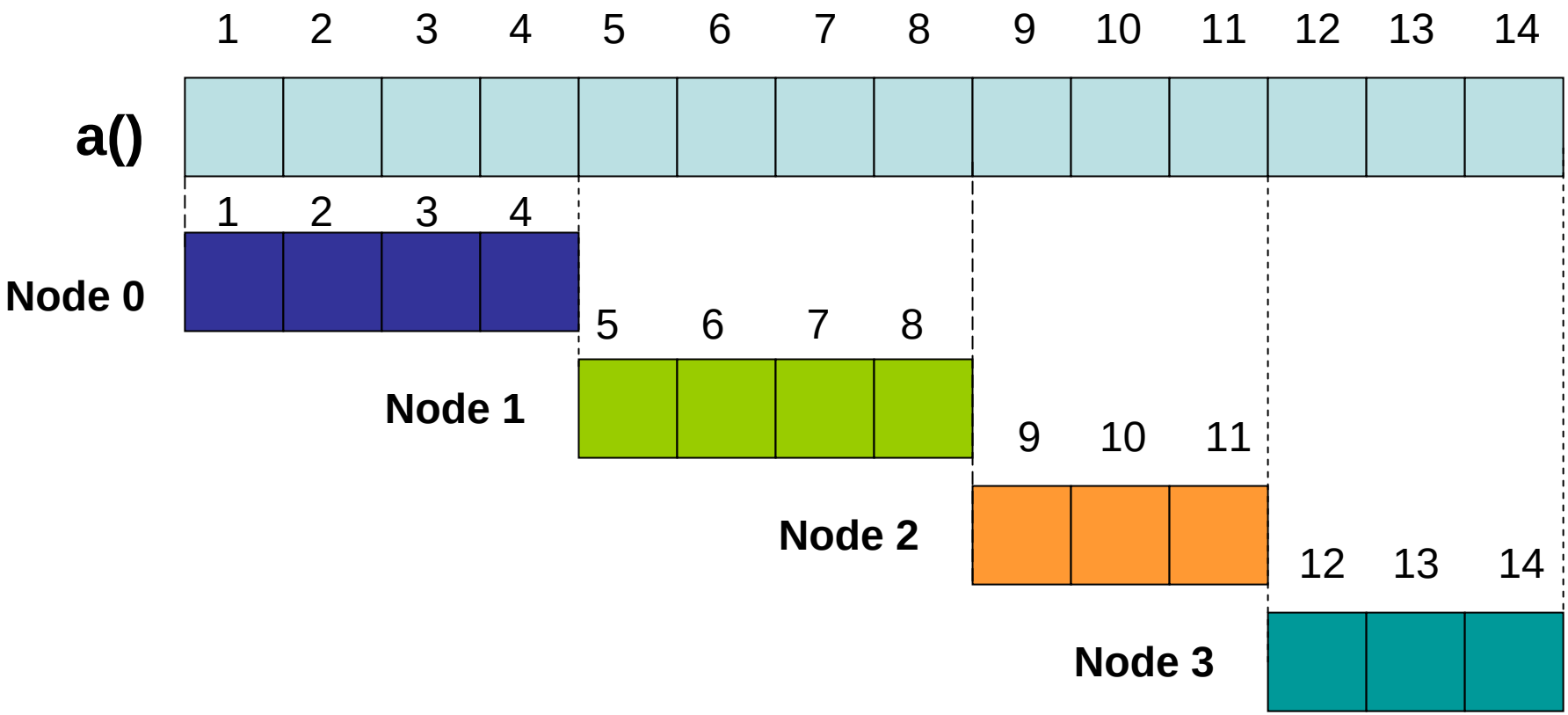
Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	1	2	3	0	1	2	3	0	1	2	3	0	1

Round and robin for summing up array elements:

```
PROGRAM main
INCLUDE 'mpif.h'
PARAMETER (n = 1000)
DIMENSION a(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i = 1 + myrank, n, nprocs
   a(i) = i
ENDDO
sum = 0.0
DO i = 1 + myrank, n, nprocs
   sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,  &
                        MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
PRINT *,'sum =',sum
CALL MPI_FINALIZE(ierr)
END program main
```

Shrinking Arrays:

Block distribution of 14 iterations over 4 cores. Each core needs only part of the array a()



```
PROGRAM main
INCLUDE 'mpif.h'
PARAMETER (n1 = 1, n2 = 1000)
REAL, ALLOCATABLE :: a(:)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(n1, n2, nprocs, myrank, ista, iend)
ALLOCATE (a(ista:iend))
DO i = ista, iend
  a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
  sum = sum + a(i)
ENDDO
DEALLOCATE (a)
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL, &
     MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
PRINT *,'sum =',sum
CALL MPI_FINALIZE(ierr)
END program main
```

Parallel Input – Output (I/O):

Traditionally, most scientific applications have been compute bound. That is, the time to completion has been dominated by the time needed to perform computation. Increasingly, however, more and more scientific applications are becoming I/O bound. That is, the time to completion is being dominated by the time required to read data from disk and/or write data to disk. Increasingly, disk I/O time is becoming the limiting factor on problem sizes that can be computed. As processor and network performance continues to increase, this problem will only worsen.

Traditional methods for writing to disk:

(1) One master process handles all I/O

This solution works fine for application with a relatively small amount of I/O required. **However, as the amount of data increases, this solution does not scale and an I/O bottleneck may develop.**

(2) Each process writes/reads its own file

This can have very good performance if the process specific files are written to disks local to each processor. However, if this output is needed for future use, it must be gathered in some way. Additionally, if the output is used for check pointing, any restart requires the same processors be used as the ones that wrote out the data.

(3) Each process accesses disk at the same time

Though it looks like all the disk access is being done at once, if the underlying file system does not support simultaneous disk access (true for traditional Unix style I/O), all the disk accesses will be sequential. This results in poor performance and a bottleneck may develop as the problem size increases.

None of these is satisfactory for large problems