An Introduction to MPI Programming

Paul Burton

Paul.Burton@ecmwf.int

Topics

- Introduction
- Basic Concepts
- Useful MPI references
- “Hello World” – the simplest MPI program
- Compiling & running on the Cray
- Synchronisation
- Sends & Receives
- Collective communications
- Reduction operations
- Blocking & non-blocking sends & receives

Introduction (1)

- Message Passing evolved in the late 1980's
- Cray was dominate in supercomputing
  - with very expensive shared-memory vector processors
  - Typically 8-16 custom made very powerful CPUs
- Many companies tried new (cheaper!) approaches to HPC
- Workstation and PC Technology was developing rapidly
  - High Volume = Cheap
- "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 commodity microprocessors (T3D/T3E)
- But application developers needed portable software

Introduction (2)

- 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 (even a heterogeneous system)
  - 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 between tasks to resolve data dependencies
- Primarily intended for communication over a network of Distributed Memory Nodes
  - But can also be used with a shared-memory node
- Can scale to thousands of processors - subject to constraints of Amdahl’s Law

Introduction (3)

- 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)
  - And many more additions in MPI version 3 (2012)
  - MPI version 1 contains the core operations, and works whatever version of MPI you have
- 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 (version 1) routines
    - Hidden within “MPL” library
    - Send/receives and some collective operations

Introduction (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

SPMD & MPMD

- The SPMD model is by far the most common
  - Single Program Multiple Data
  - The same executable runs multiple times simultaneously on different processors
  - The problem is divided across the multiple executables
  - Each executable works on a subset of the data
- MPMD
  - Multi Program Multiple Data
  - Different executable on different processors
  - Useful for coupled models for example
    - eg. atmosphere executable, ocean executable, coupling executable
  - Part of the MPI 2 standard
  - Not currently used by IFS
  - Can be mimicked in SPMD mode with a single executable
    - Top level branch deciding which “program” (subroutine) this task will run

Some definitions

- Task
  - one running instance (copy) of a program – the basic unit of an MPI parallel execution
  - Equivalent to a UNIX process
  - Each task has direct access to its own memory, but not that of other tasks
  - May run on one processor
    - Or across many if OpenMP is used as well (threads)
    - Or many tasks on one processor (not a good idea!)
- Master
  - the master task is by convention, usually 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

Useful MPI references

- MPI standard
  - Lots of useful information about MPI’s behaviour & implementation
  - http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html
- Open MPI documentation
  - A nice easy to use guide to the API (contains MPI v2 too), including Fortran interface
  - http://www.open-mpi.org/doc/v1.10/
- MPI tutorials
  - https://computing.llnl.gov/tutorials/mpi/
  - http://mpitutorial.com/tutorials/

“Hello world” MPI program

- Basic components in all MPI programs
  - Four 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 using MPI
- 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
  - avoid doing 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

```
use mpi
call MPI_COMM_SIZE(MPI_COMM_WORLD,...
```

- An MPI communicator
  - A communicator defines a set or group of MPI tasks
- Constant integer value from “`use mpi`”
- `MPI_COMM_WORLD` means all tasks
  - many MPI programs only ever use `MPI_COMM_WORLD`
  - All our examples only use `MPI_COMM_WORLD`
- You can create your own communicators to define subsets of MPI tasks
  - 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)

“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 MPI tasks in the given communicator
  - `MPI_COMM_WORLD` in this case – so it's the total number of MPI tasks
  - Value is returned in variable "`ntasks`"
  - The total number of MPI tasks is set from the environment in which you launched the parallel executable
    - eg. `aprun` on the Cray
- Value can be used to help decompose the problem
  - The size of a local array will often be a function of the total data size and the number of MPI tasks to split the data over

“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,ntasks,mytask
call MPI_COMM_RANK(MPI_COMM_WORLD, mytask, ierror)
```

- Returns the rank (location) of this task within the communicator supplied
  - Returns the rank in variable “`mytask`”
- In the range `0` to `ntasks-1` (for the MPI_COMM_WORLD communicator group)
  - Used as a task identifier when sending/receiving messages
  - WARNING : Easy to make mistakes with this as Fortran arrays normally run 1:n

“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
  - Technically it should be only the tasks in the defined communicator
  - All known implementations abort all the tasks

- Even if only one task makes call

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
       Followed by
$ 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 (where your interactive shell is running), launches the parallel executable on the parallel (ESM) node(s)
  - If you're not in queue "np" (parallel job), then aprun isn't available…

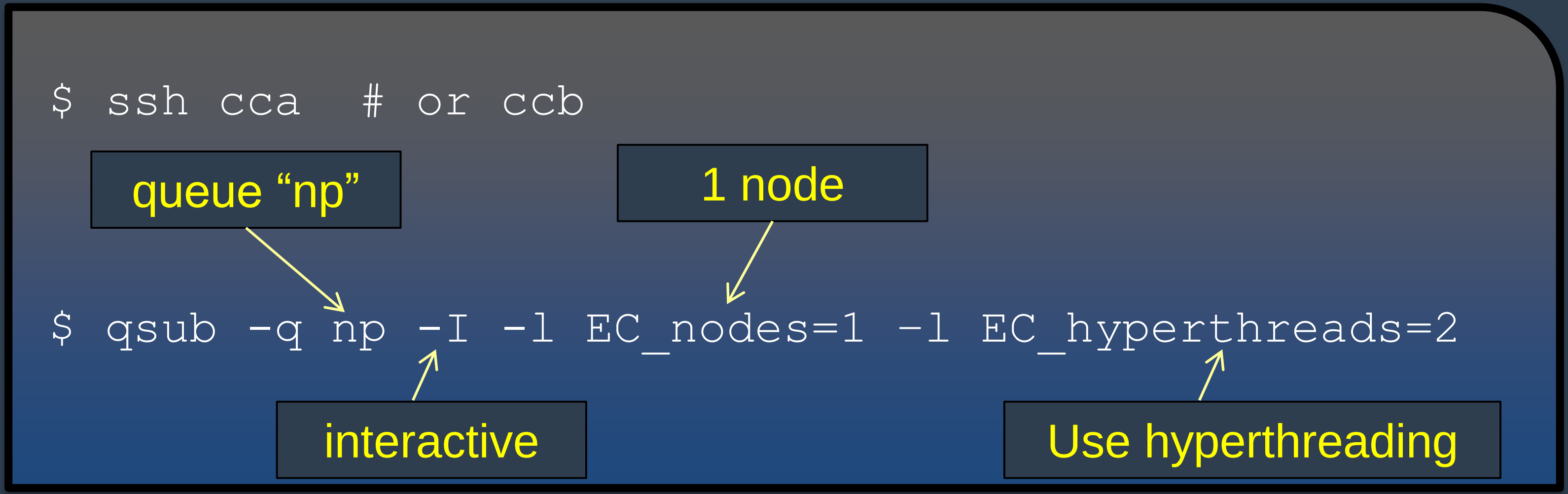
```
$ aprun -n 4 <executable>
```

- `mpiexec`
  - Equivalent command in "nf" (fraction job) or "ns" (serial job) queue

```
$ module load cray-snplauncher
$ mpiexec -n 4 <executable>
```

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 (72 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

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

Practical 1

- Copy all the practical exercises to your account on cca or ccb:

```
$ ssh cca # or ccb

$ mkdir mpi_course ; cd mpi_course

$ cp -r ~trx/mpi.2017/* .
```

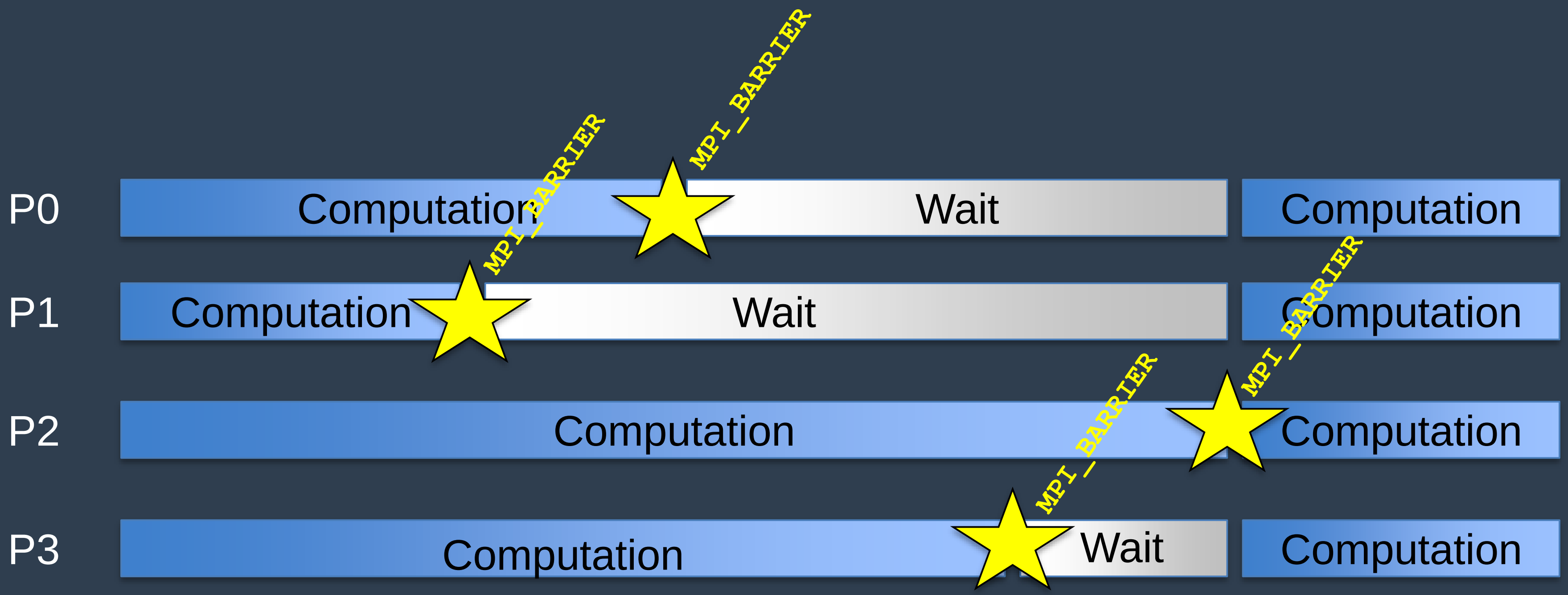
- Exercise1a
  - Run your own “Hello World” program with MPI

- See the README for details

MPI_BARRIER

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

- Forces all tasks in the specified communicator group to synchronise (wait for each other)



MPI_BARRIER

- A task waits in the barrier until every task has reached it
- Then all tasks exit the call together at the same time
- Deadlock if one task does not reach the barrier
  - `MPI_BARRIER` will wait until the task reaches its cpu limit
- What happens if different tasks call `MPI_BARRIER` in different parts of the code?
  - Could be desired behaviour, or it could be highly confusing bug!

- Why do we need a `MPI_BARRIER`?
  - To ensure a computation is complete before we do some communications
    - Although most communications allow us to “block” to do a synchronisation only between the processors involved
  - To do timing
    - Allows us to measure the time taken by the “slowest” task
  - To enforce an ordering of operations

Enforcing an ordered output using `MPI_BARRIER`

```
WRITE(6,*) 'Some information from task ',MYPROC
```

- What order will these outputs appear in from the different MPI tasks?
- How can we enforce an ordering?
- Where could we add an MPI_BARRIER to force an ordered output?

```
DO proc=1,MYPROC
  IF (MYPROC == proc) THEN
    WRITE(6,*) 'Some information from task ',MYPROC
  ENDIF
ENDDO
```

Practical 2

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

Message Passing : SEND and RECEIVE

- `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 – the body and some headers
    - To: Where the message should be sent to (in MPI, the receiving TaskID)
    - Subject: Some description of the contents (in MPI, a “tag”)
    - Body: The data itself (can be any size), all basic 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 not covered here

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 (rather like an email's "subject")
- Particularly useful when sending multiple messages
  - You can chose to receive the particular message you're interested in by filtering for a particular tag
- You decide what tag values to use
  - Good idea (helps spot problems) to use separate ranges of tags in different communication areas, eg:
    - 1000, 1001, 1002..... in routine a
    - 2000, 2001, 2002.... in routine b
  - Prevents inadvertent communication between "unmatched" SENDs and RECEIVESs

MPI_SEND

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

Argument	Description	Intent
SBUF	The array being sent	Input
COUNT	The number of elements to send	Input
MPI_TYPE	Type of `SBUF` (eg. `MPI_REAL`) *These type descriptions come from "`use mpi`"*	Input
DEST	The taskID to send the message to *TaskID is the rank of the task within the communicator*	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)
```

Argument	Description	Intent
RBUF	The array being received	Output
COUNT	The length of RBUF	Input
MPI_TYPE	Type of RBUF (eg. MPI_REAL)	Input
SOURCE	The taskID 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
- What order will messages be received in?
  - 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_RECVs` in the same order as the `MPI_SENDs` (ie. matching tags)
  - Otherwise the first `MPI_RECV` will wait forever, and eventually die
  - *What happens if you don't know the ordering of the `MPI_SENDs`?*

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
    - If different messages will be different lengths – we need to ensure the “rbuf” array is big enough for the longest message
- 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

An example : task 0 sends a message to task 1

```fortran
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)

- `MPI_SEND/MPI_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 (2)

- MPI contains many Collective Communications routines
  - called by all tasks (in a communicator group) together
  - replace multiple send/receive calls
  - easier to code and understand
  - can be more efficient
  - the MPI library may optimise the data transfers
- We will look at a small subset of some of the more common colllectives
- 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

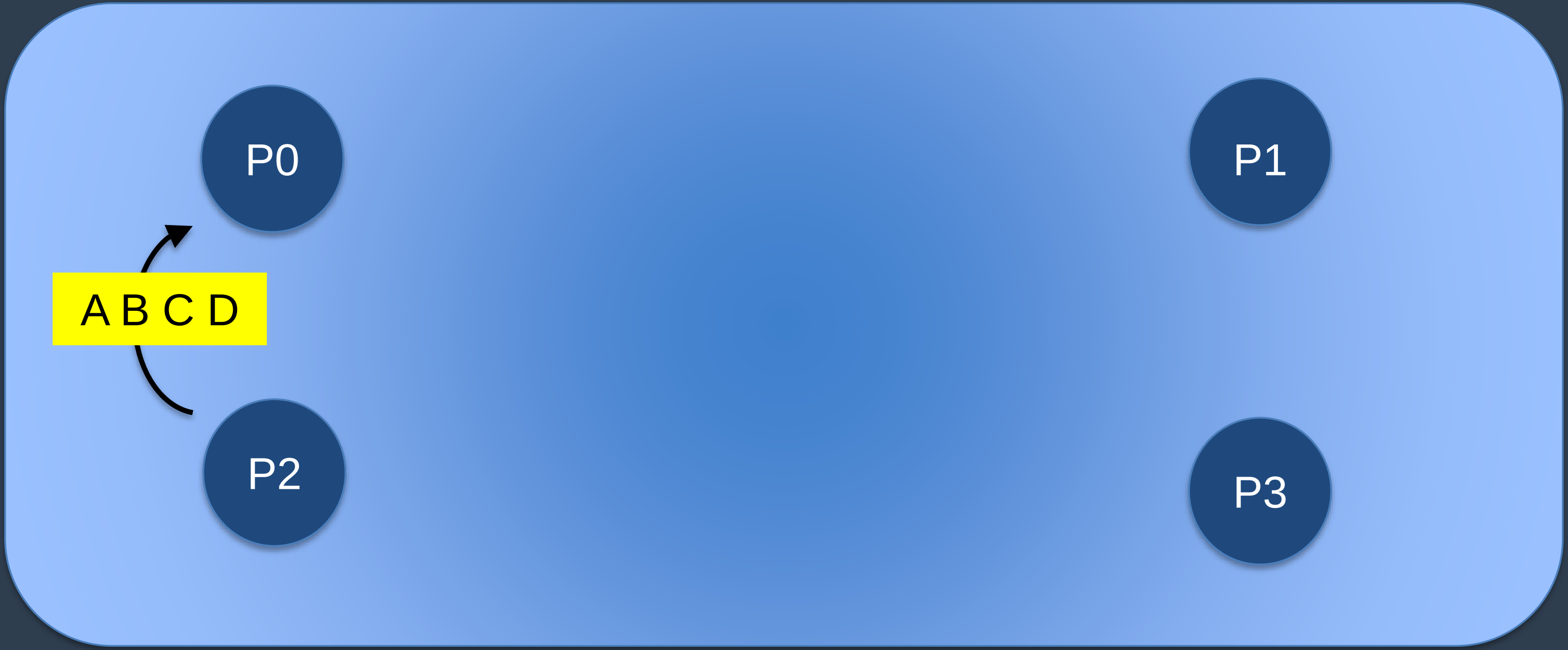
MPI_BCAST

P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\text{MPI\_BCAST}}$

P0				
P1				
P2				
P3				

MPI_BCAST

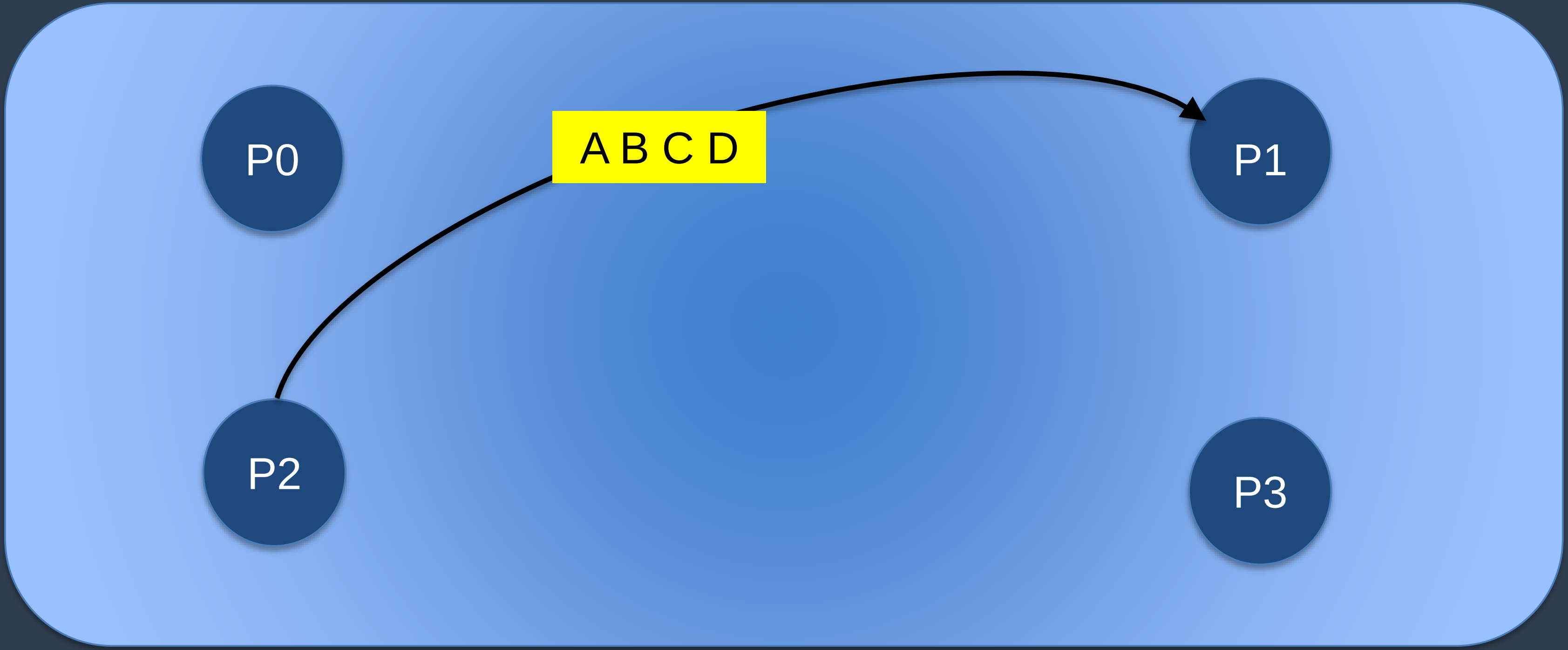


P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\texttt{MPI\_BCAST}}$

P0	A	B	C	D
P1				
P2				
P3				

MPI_BCAST

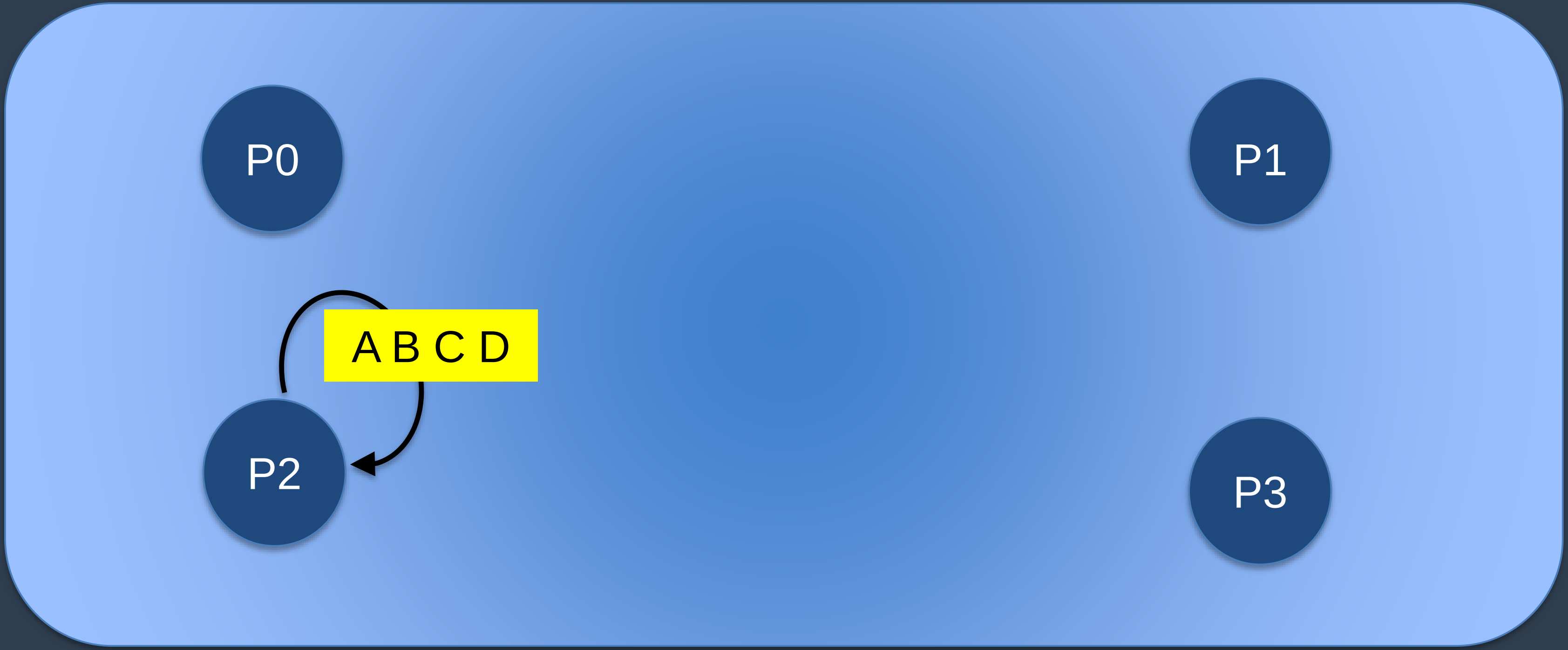


P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\texttt{MPI\_BCAST}}$

P0	A	B	C	D
P1	A	B	C	D
P2				
P3				

MPI_BCAST

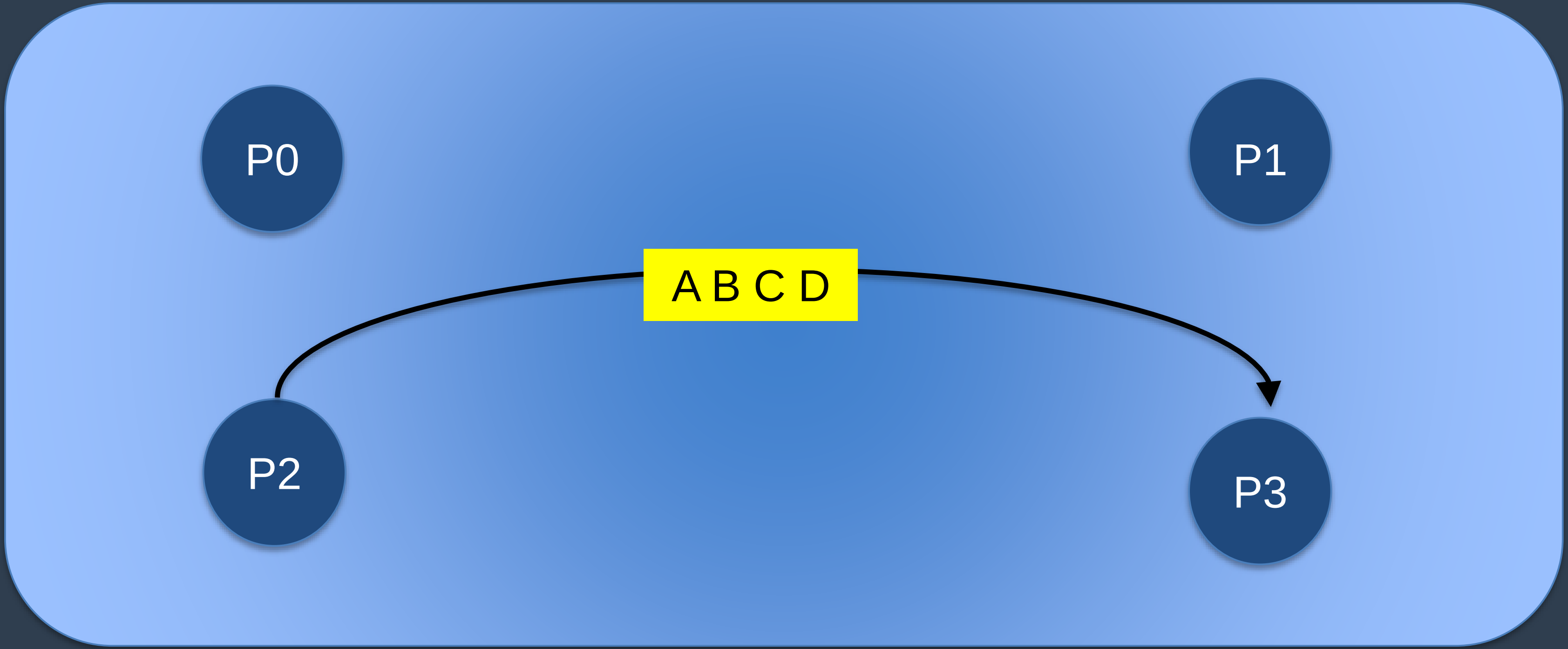


P0				
P1				
P2	A	B	C	D
P3				

MPI_BCAST →

P0	A	B	C	D
P1	A	B	C	D
P2	A	B	C	D
P3				

MPI_BCAST



P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\texttt{MPI\_BCAST}}$

P0	A	B	C	D
P1	A	B	C	D
P2	A	B	C	D
P3	A	B	C	D

MPI_BCAST

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

Argument	Description	Intent
BUFF	The array being broadcast	Input/Output
COUNT	The number of elements to broadcast	Input
MPI_TYPE	Type of `BUFF` (eg. `MPI_REAL`)	Input
ROOT	The taskID doing the broadcast	Input

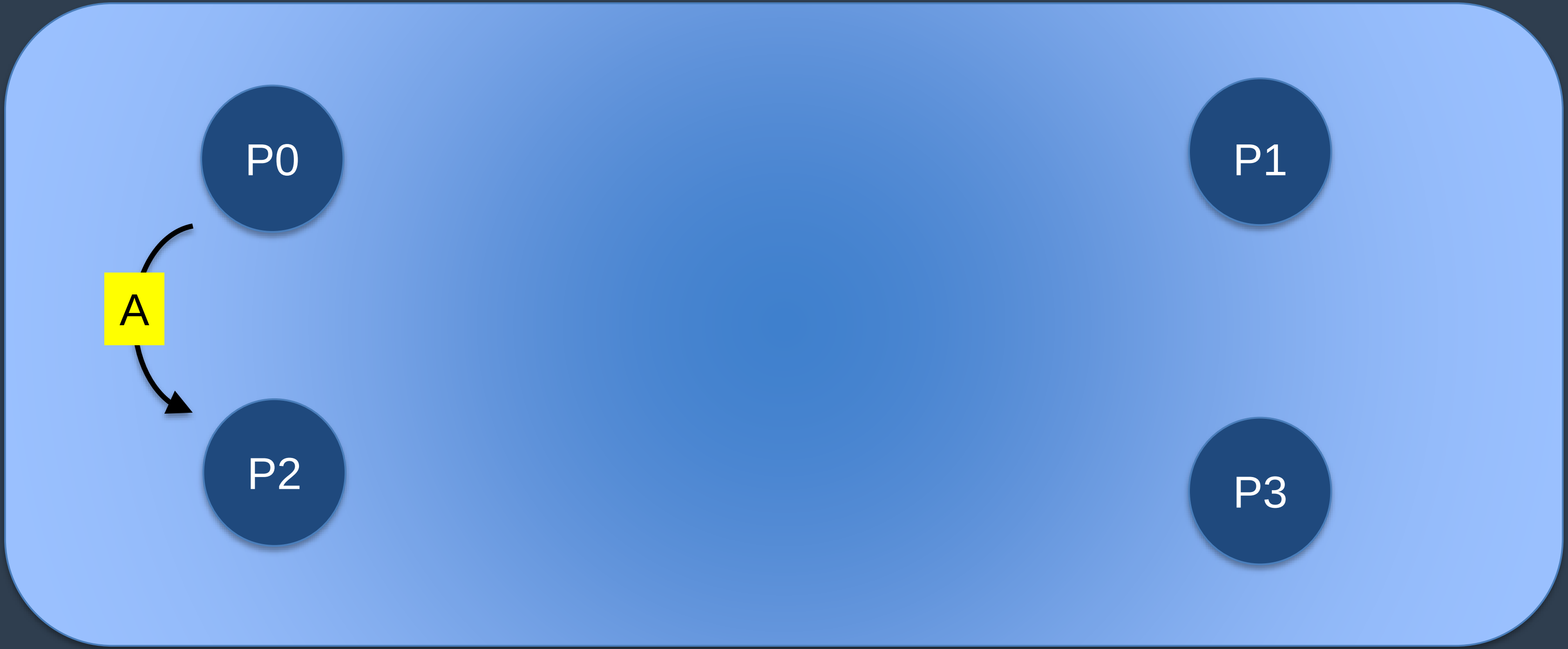
MPI_GATHER

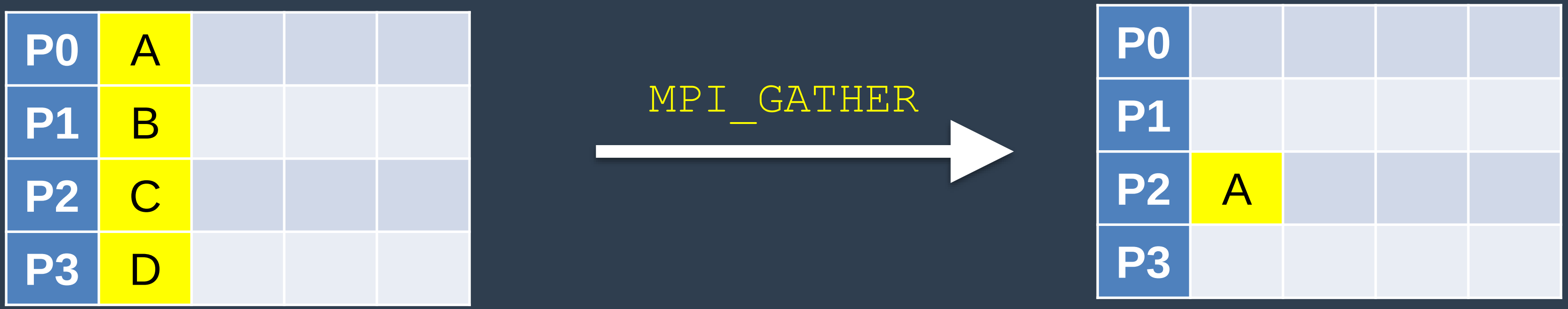
P0	A			
P1	B			
P2	C			
P3	D			

MPI_GATHER →

P0				
P1				
P2				
P3				

MPI_GATHER

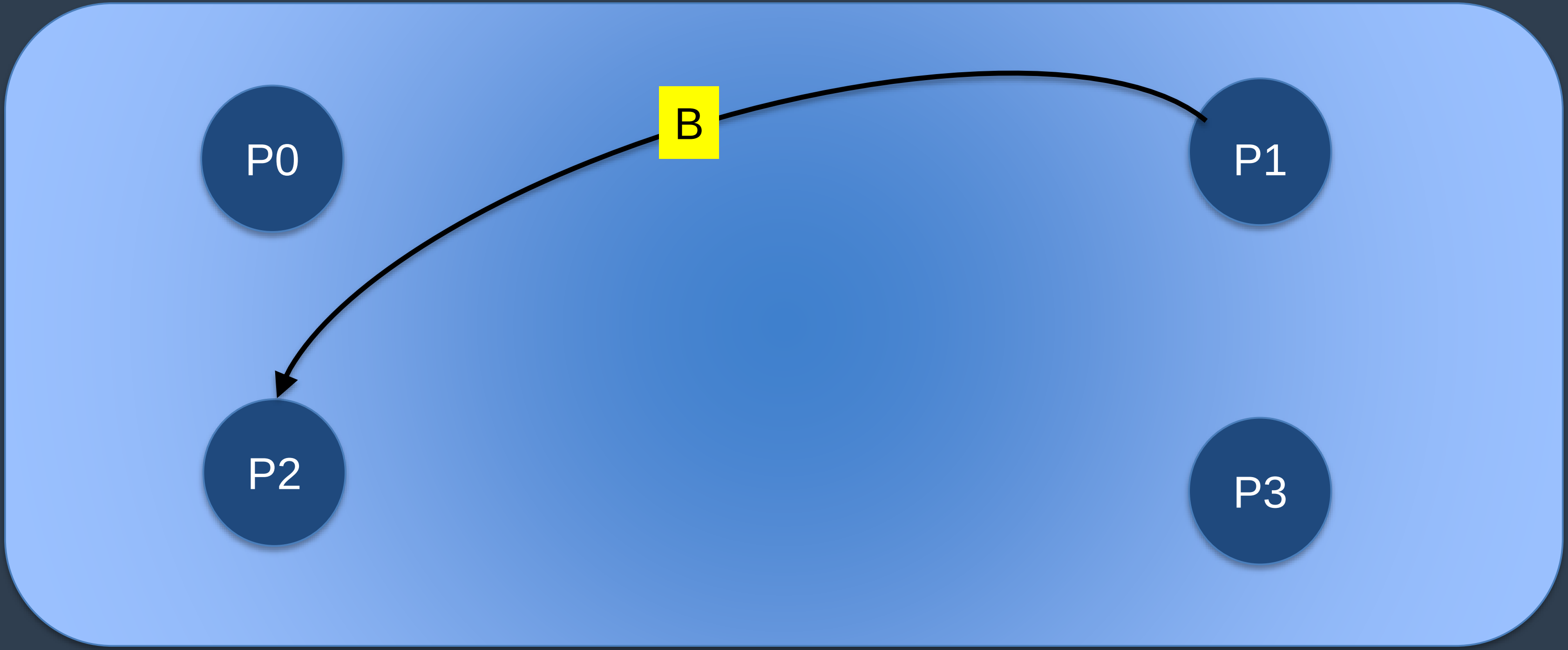




P0	A			
P1	B			
P2	C			
P3	D			

P0				
P1				
P2	A			
P3				

MPI_GATHER

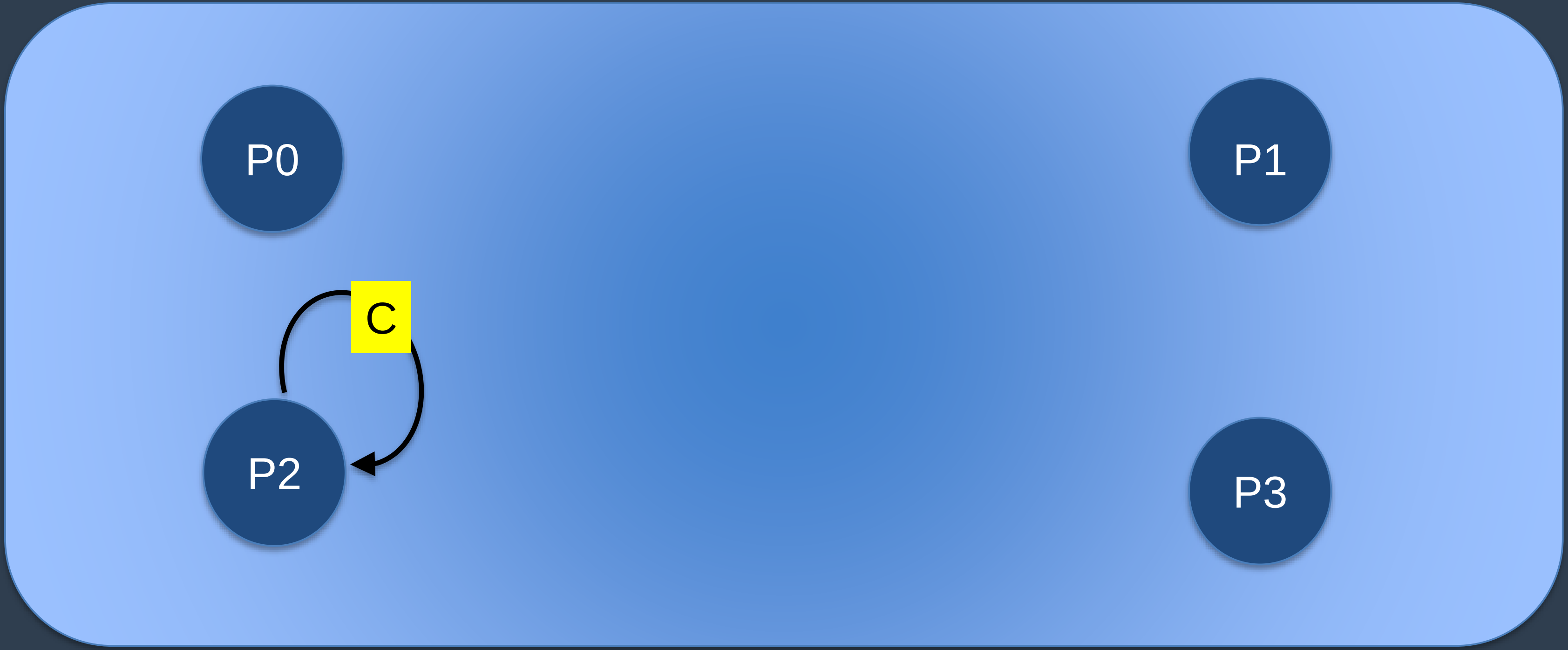


P0	A			
P1	B			
P2	C			
P3	D			

MPI_GATHER →

P0				
P1				
P2	A	B		
P3				

MPI_GATHER

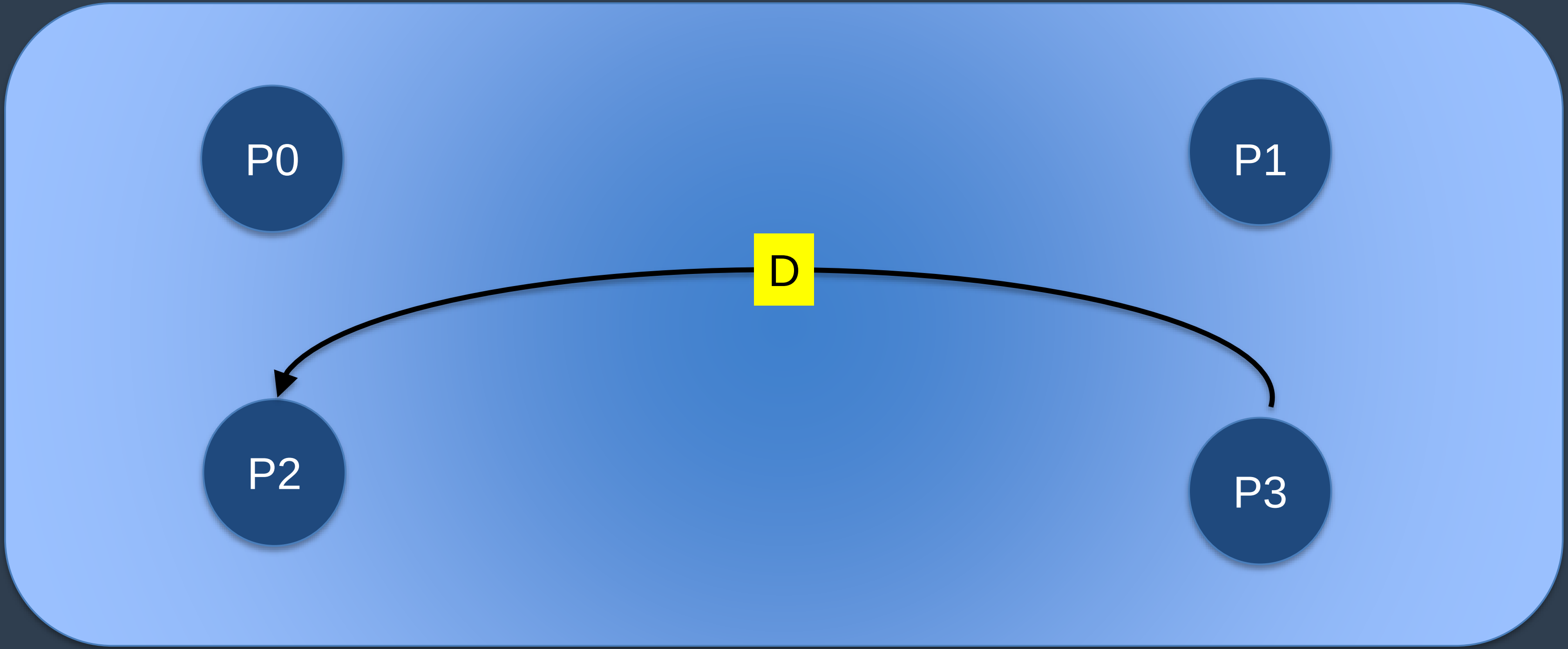


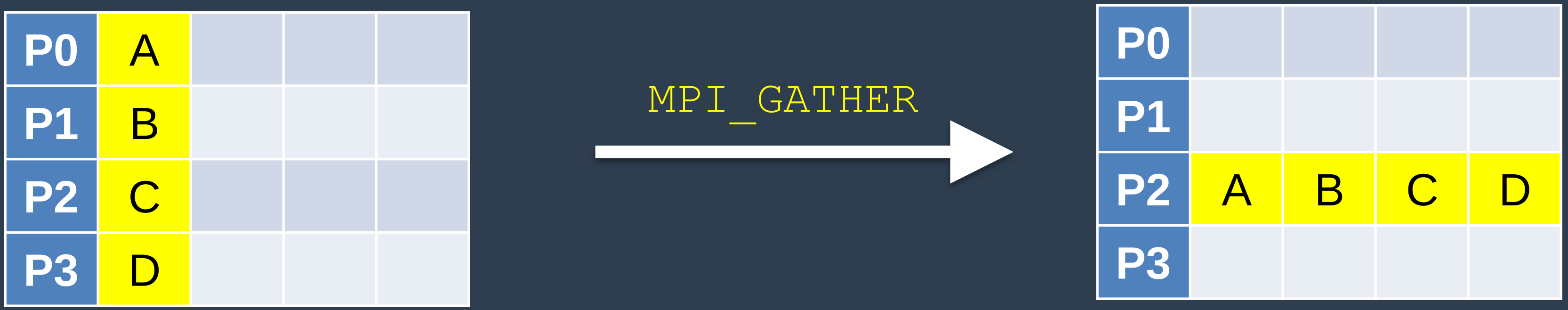
P0	A			
P1	B			
P2	C			
P3	D			

MPI_GATHER →

P0				
P1				
P2	A	B	C	
P3				

MPI_GATHER





P0	A			
P1	B			
P2	C			
P3	D			

P0				
P1				
P2	A	B	C	D
P3				

MPI_GATHER

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

Argument	Description	Intent
SBUFF	The array being sent	Input
SCOUNT	Number of items being sent	Input
SEND_TYPE	Type of `SBUFF` (eg. `MPI_REAL`)	Input
RBUFF	The array being received	Output
RCOUNT	The number of elements to receive	Input
RECEIVE_TYPE	Type of `SBUFF` (eg. `MPI_REAL`)	Input
ROOT	The taskID doing the gather	Input

A few variants on `MPI_GATHER`

- `MPI_ALLGATHER`
  - gather arrays of equal length into one array on <u>all</u> tasks
  - Equivalent to doing `MPI_GATHER` followed by `MPI_BCAST`
  - or doing a `MPI_BCAST` from each task

- `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?

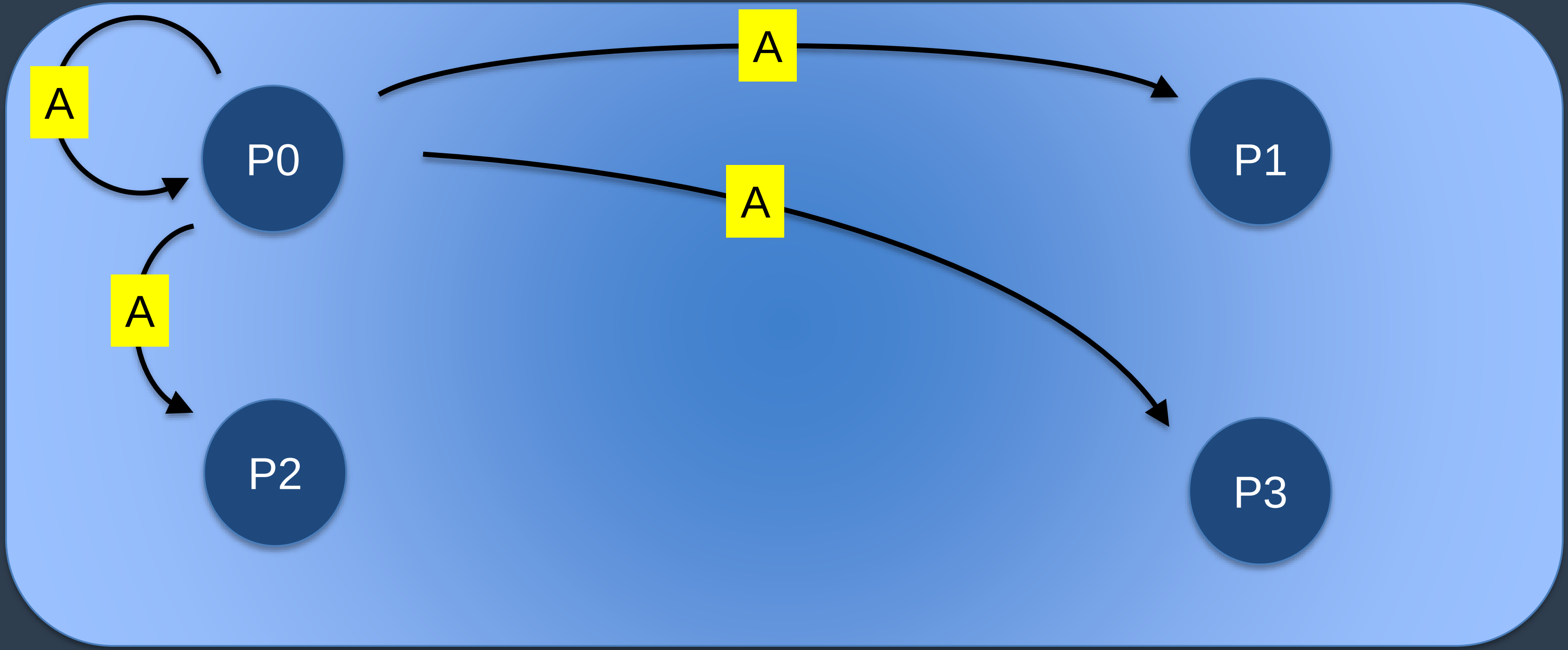
MPI_ALLGATHER

P0	A			
P1	B			
P2	C			
P3	D			

$\xrightarrow{\texttt{MPI\_ALLGATHER}}$

P0				
P1				
P2				
P3				

MPI_ALLGATHER

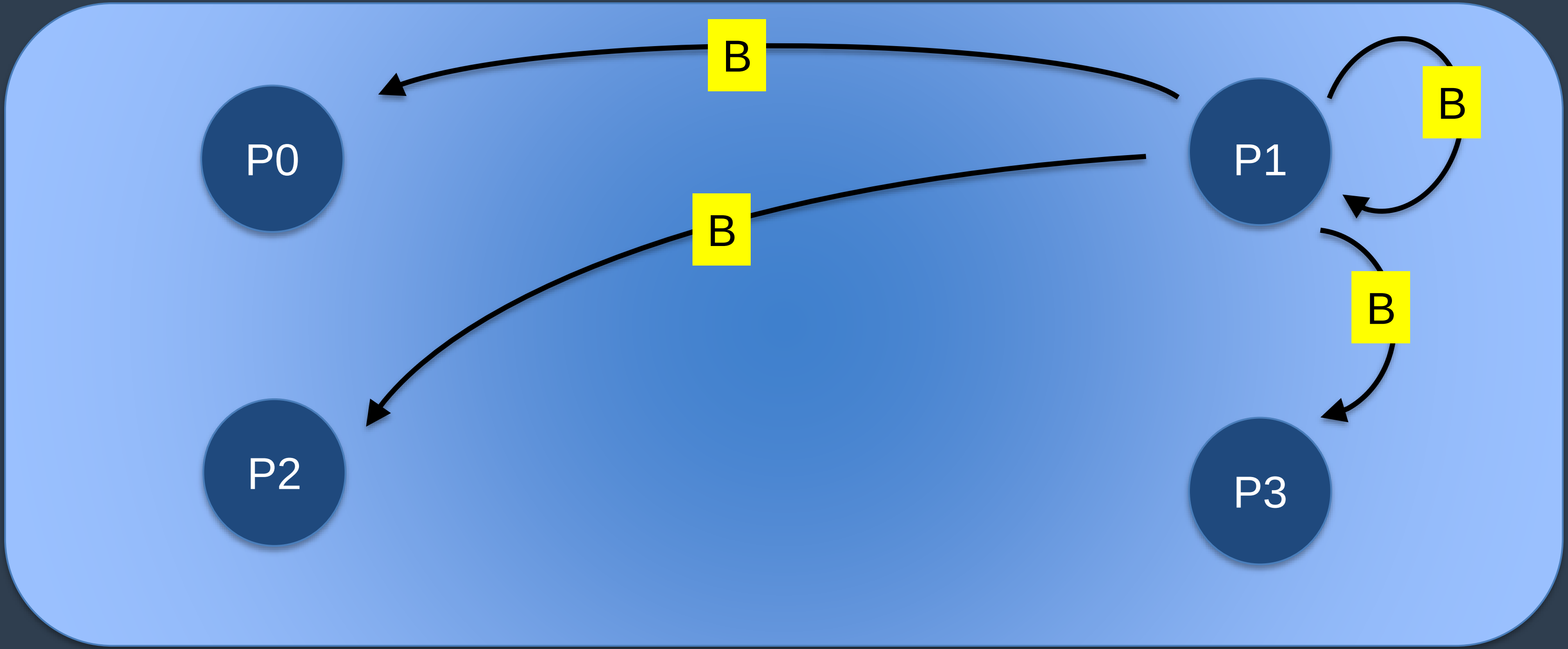


P0	A			
P1	B			
P2	C			
P3	D			

$\xrightarrow{\text{MPI\_ALLGATHER}}$

P0	A			
P1	A			
P2	A			
P3	A			

MPI_ALLGATHER

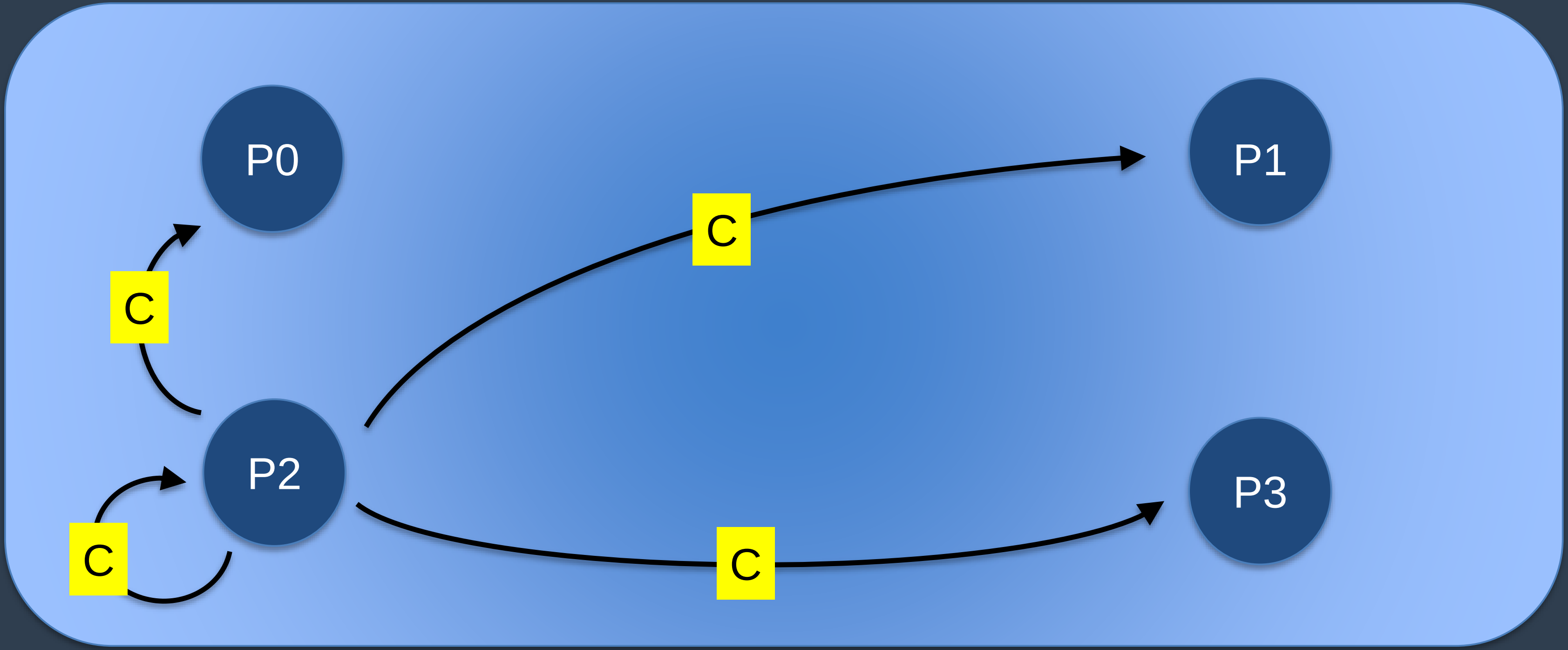


P0	A			
P1	B			
P2	C			
P3	D			

$\xrightarrow{\texttt{MPI\_ALLGATHER}}$

P0	A	B		
P1	A	B		
P2	A	B		
P3	A	B		

MPI_ALLGATHER

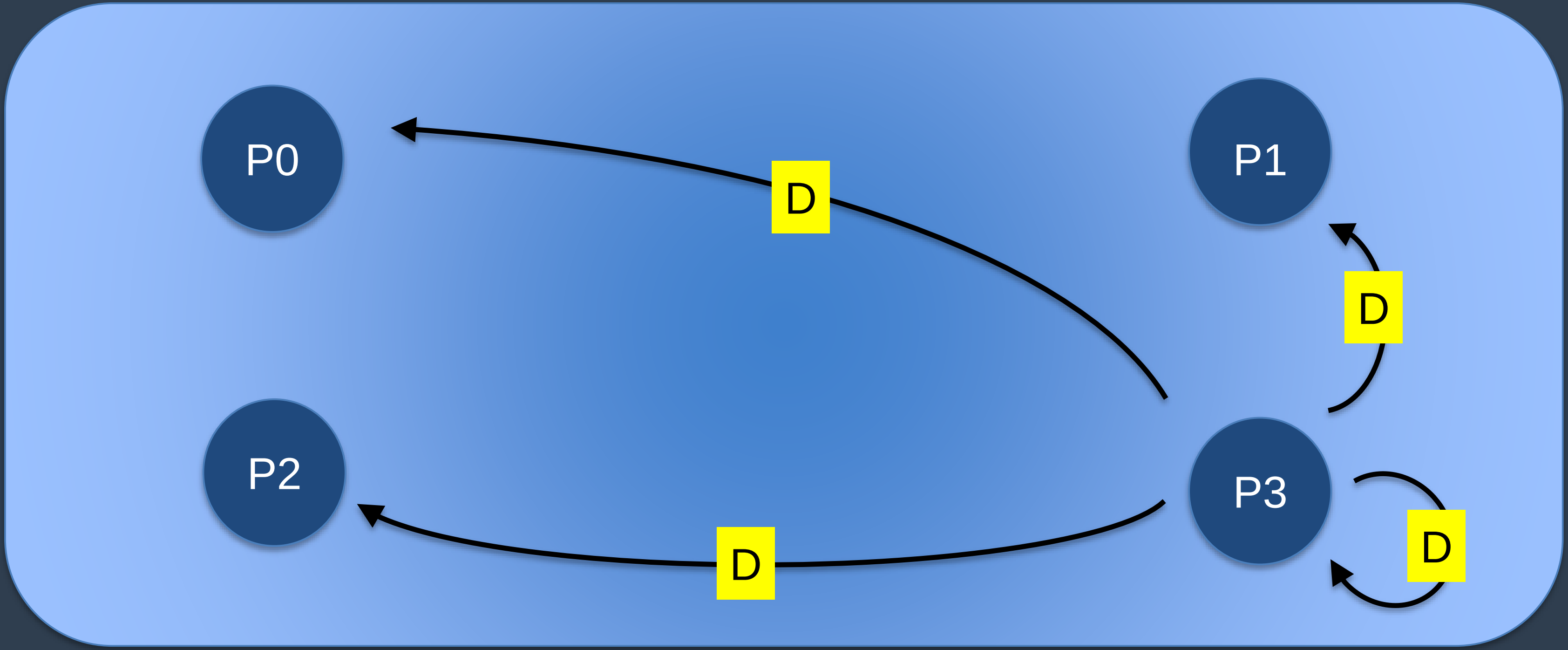


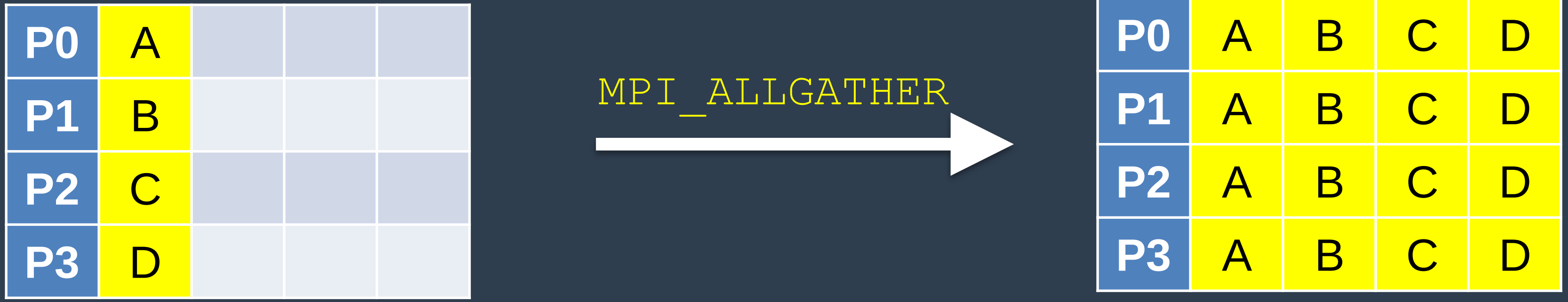
P0	A			
P1	B			
P2	C			
P3	D			

$\xrightarrow{\text{MPI\_ALLGATHER}}$

P0	A	B	C	
P1	A	B	C	
P2	A	B	C	
P3	A	B	C	

MPI_ALLGATHER





P0	A			
P1	B			
P2	C			
P3	D			

$\xrightarrow{\texttt{MPI\_ALLGATHER}}$

P0	A	B	C	D
P1	A	B	C	D
P2	A	B	C	D
P3	A	B	C	D

MPI_ALLGATHER

```
FORTRAN_TYPE:: sbuff,rbuff
integer:: count, root, ierror
call MPI_ALLGATHER( sbuff, scount, send_type,       &
                    rbuff, rcount, receive_type,    &
                    MPI_COMM_WORLD, ierror)
```

Argument	Description	Intent
SBUFF	The array being sent	Input
SCOUNT	Number of items being sent	Input
SEND_TYPE	Type of `SBUFF` (eg. `MPI_REAL`)	Input
RBUFF	The array being received	Output
RCOUNT	The number of elements to receive	Input
RECEIVE_TYPE	Type of `SBUFF` (eg. `MPI_REAL`)	Input

Scatter routines

- `MPI_SCATTER`
  - divide one array on one task <u>equally</u> amongst all tasks
  - each task receives the same amount of data
  - Equivalent putting MPI_SEND in a loop over all tasks

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

- Where do you think they might be useful?

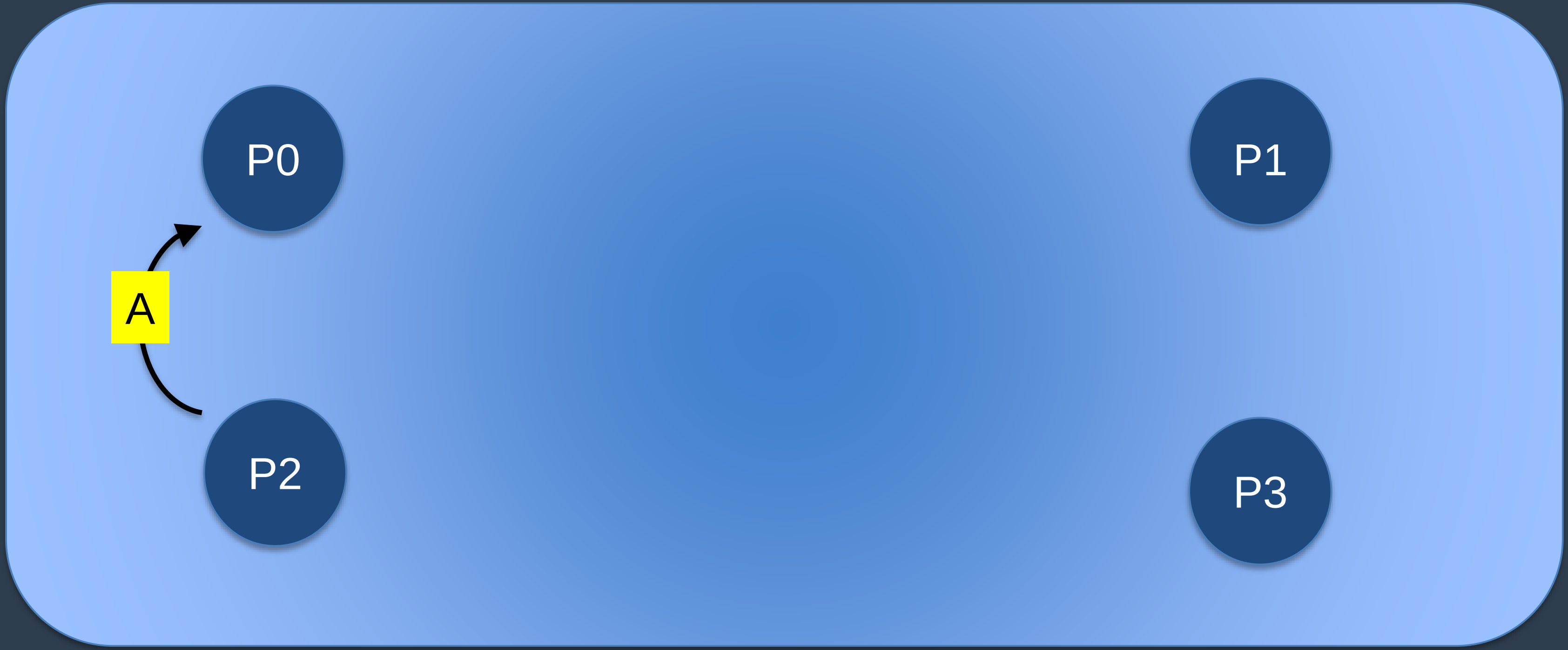
MPI_SCATTER

P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\text{MPI\_SCATTER}}$

P0				
P1				
P2				
P3				

MPI_SCATTER

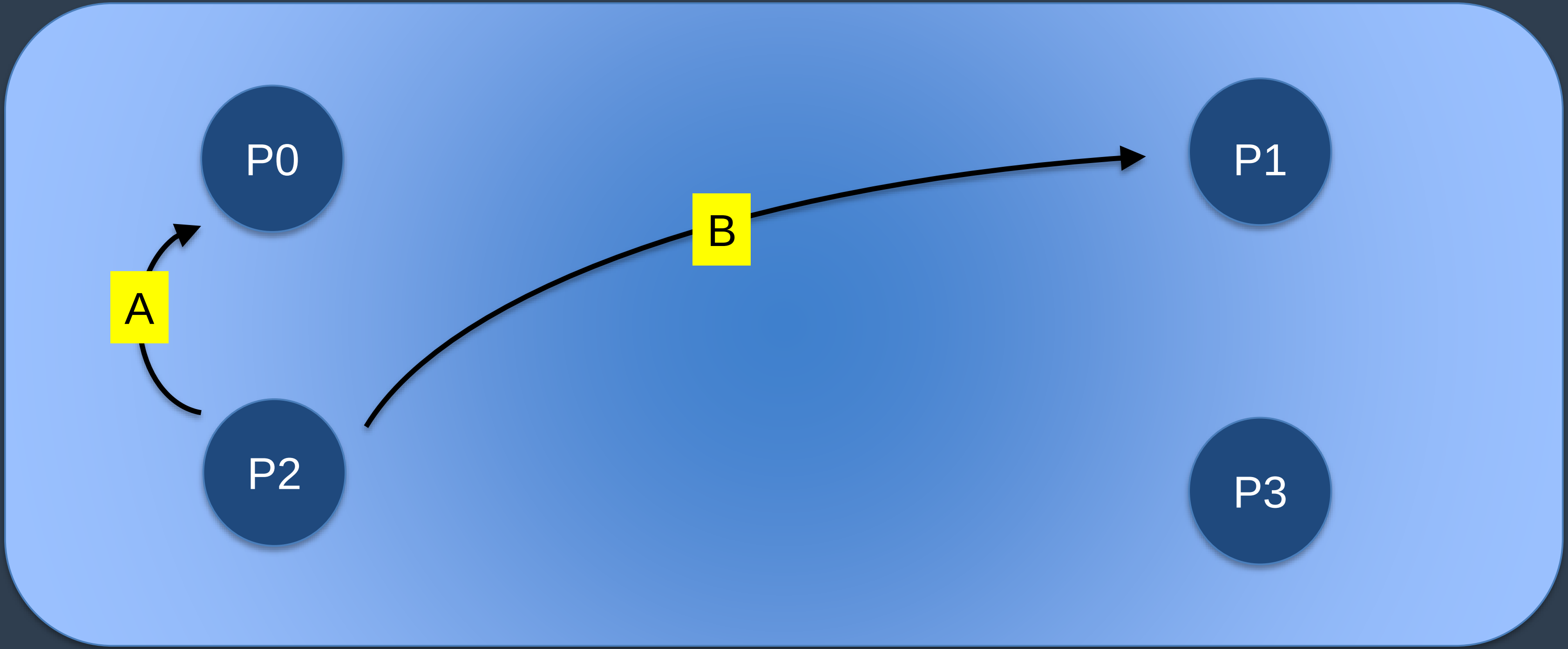


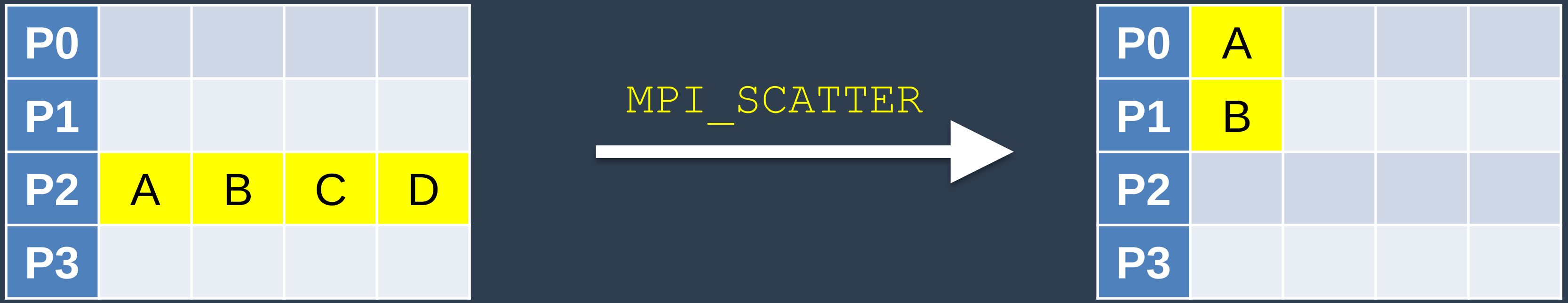
P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\text{MPI\_SCATTER}}$

P0	A			
P1				
P2				
P3				

MPI_SCATTER

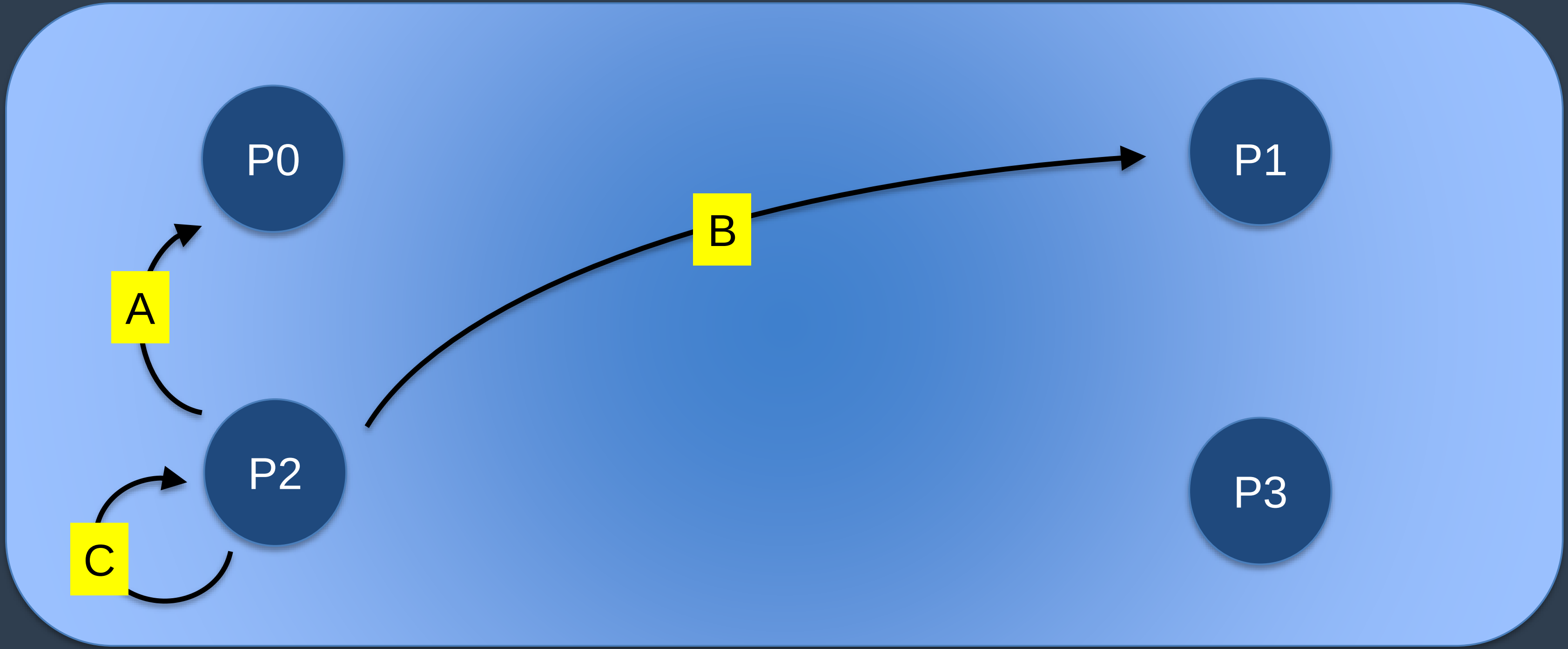




P0				
P1				
P2	A	B	C	D
P3				

P0	A			
P1	B			
P2				
P3				

MPI_SCATTER

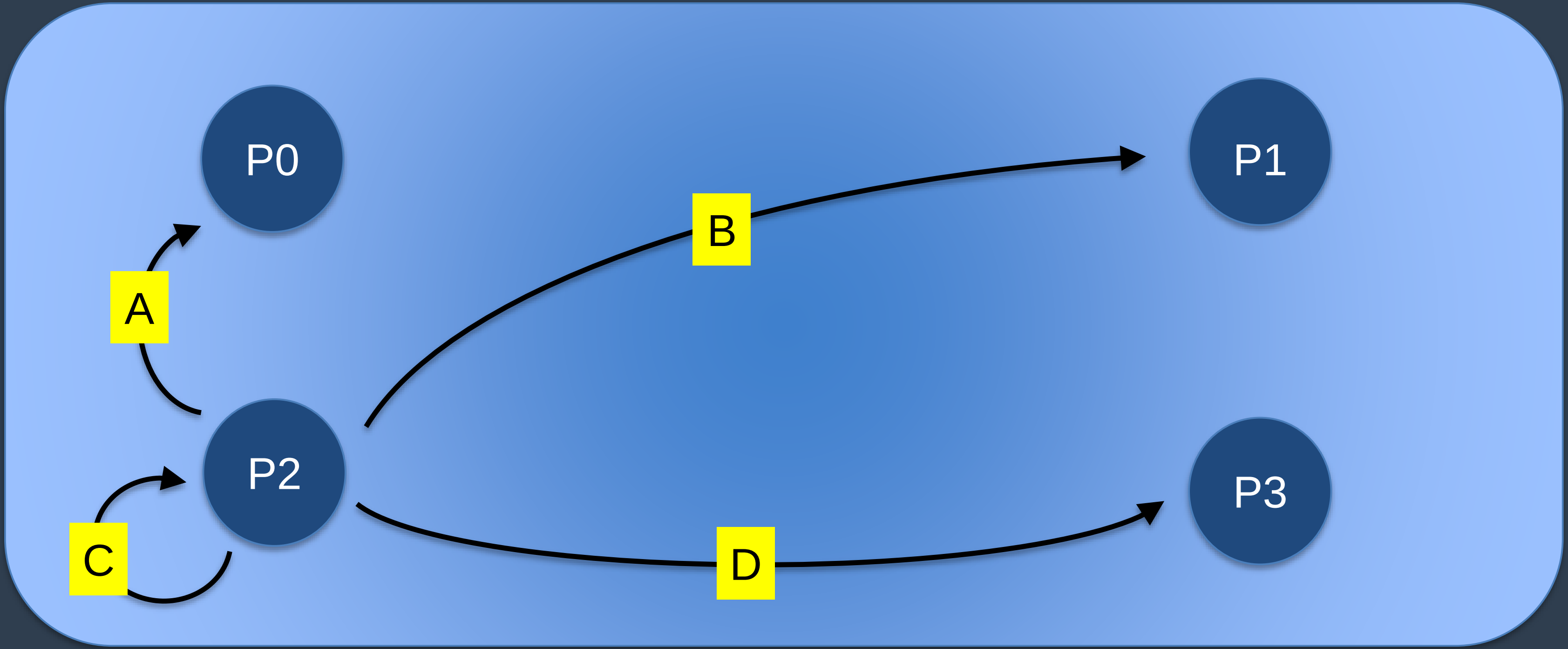


P0				
P1				
P2	A	B	C	D
P3				

$\xrightarrow{\texttt{MPI\_SCATTER}}$

P0	A			
P1	B			
P2	C			
P3				

MPI_SCATTER



P0				
P1				
P2	A	B	C	D
P3				

P0	A			
P1	B			
P2	C			
P3	D			

MPI_SCATTER

```
FORTRAN_TYPE:: sbuff,rbuff

integer:: count, root, ierror

call MPI_SCATTER( sbuff, scount, send_type,      &
                  rbuff, rcount, receive_type,   &
                  root,MPI_COMM_WORLD, ierror)
```

Argument	Description	Intent
SBUFF	The array being sent	Input
SCOUNT	Number of items being sent	Input
SEND_TYPE	Type of `SBUFF` (eg. `MPI_REAL`)	Input
RBUFF	The array being received	Output
RCOUNT	The number of elements to receive	Input
RECEIVE_TYPE	Type of `SBUFF` (eg. `MPI_REAL`)	Input
ROOT	The taskID doing the gather	Input

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
  - Equivalent to putting `MPI_SEND/MPI_RECV` in a loop

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

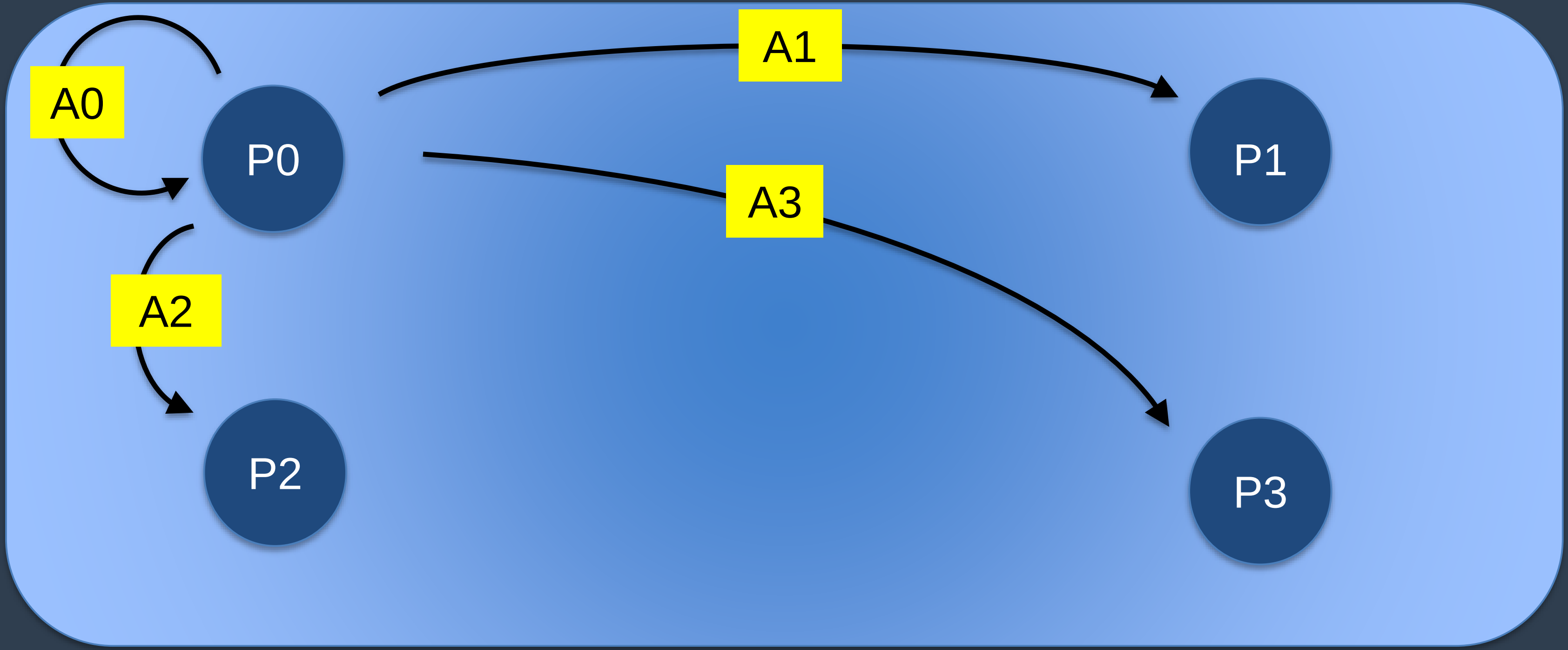
MPI_ALLTOALL

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

$\xrightarrow{\texttt{MPI\_ALLTOALL}}$

P0				
P1				
P2				
P3				

MPI_ALLTOALL

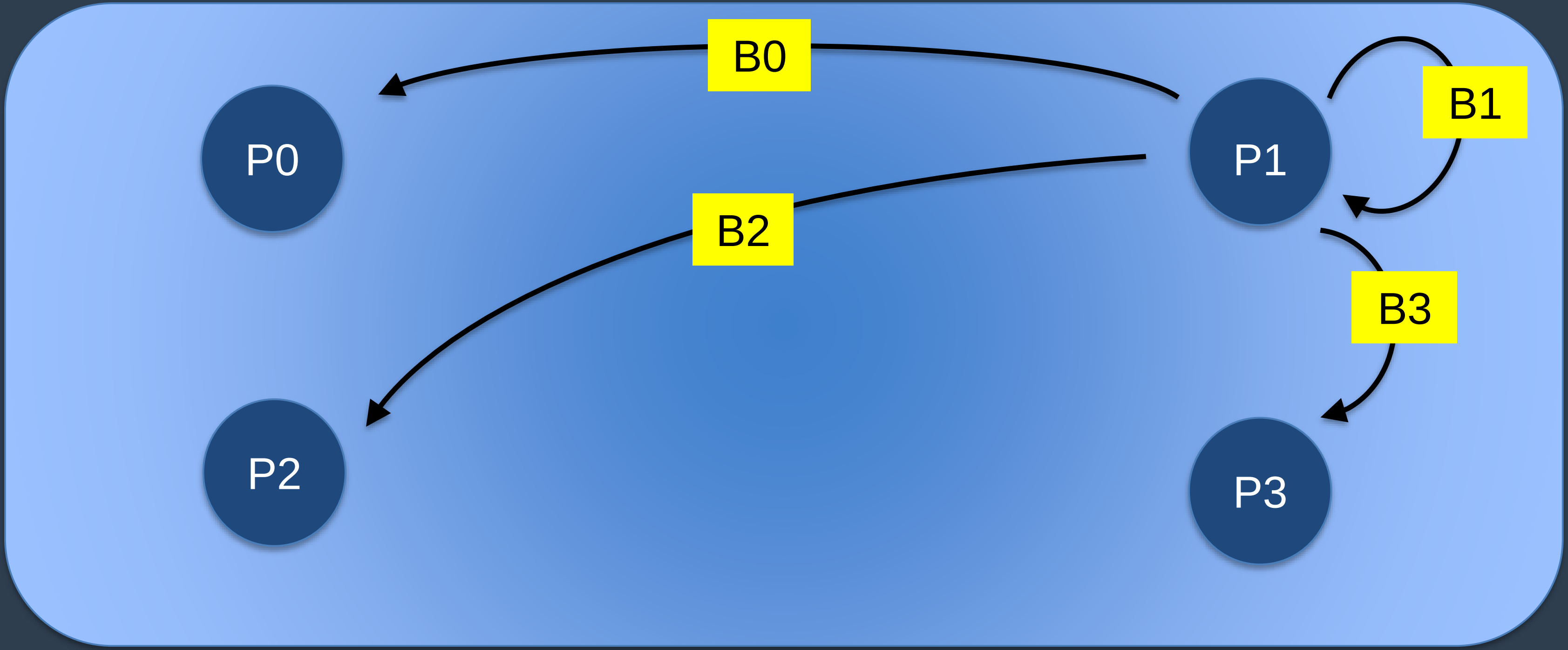


P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

$\xrightarrow{\texttt{MPI\_ALLTOALL}}$

P0	A0			
P1	A1			
P2	A2			
P3	A3			

MPI_ALLTOALL

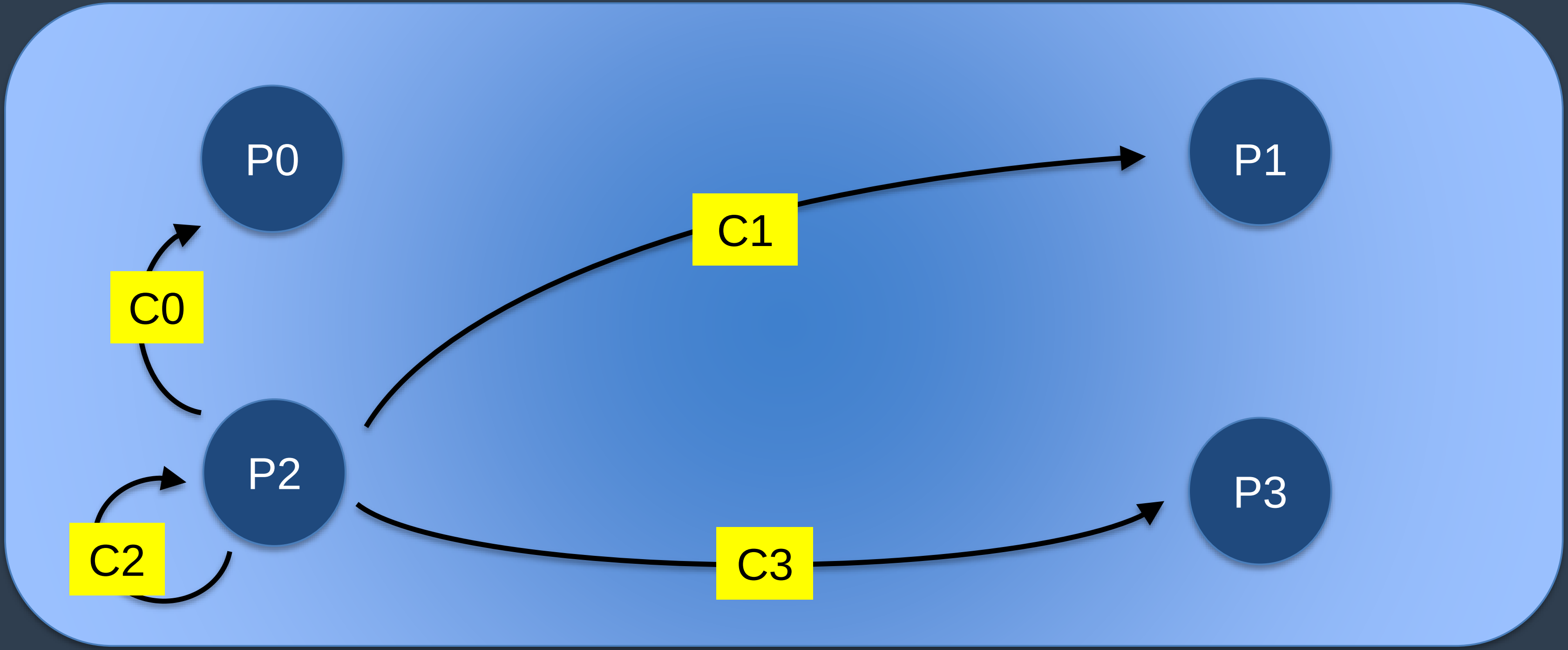


P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

$\xrightarrow{\texttt{MPI\_ALLTOALL}}$

P0	A0	B0		
P1	A1	B1		
P2	A2	B2		
P3	A3	B3		

MPI_ALLTOALL

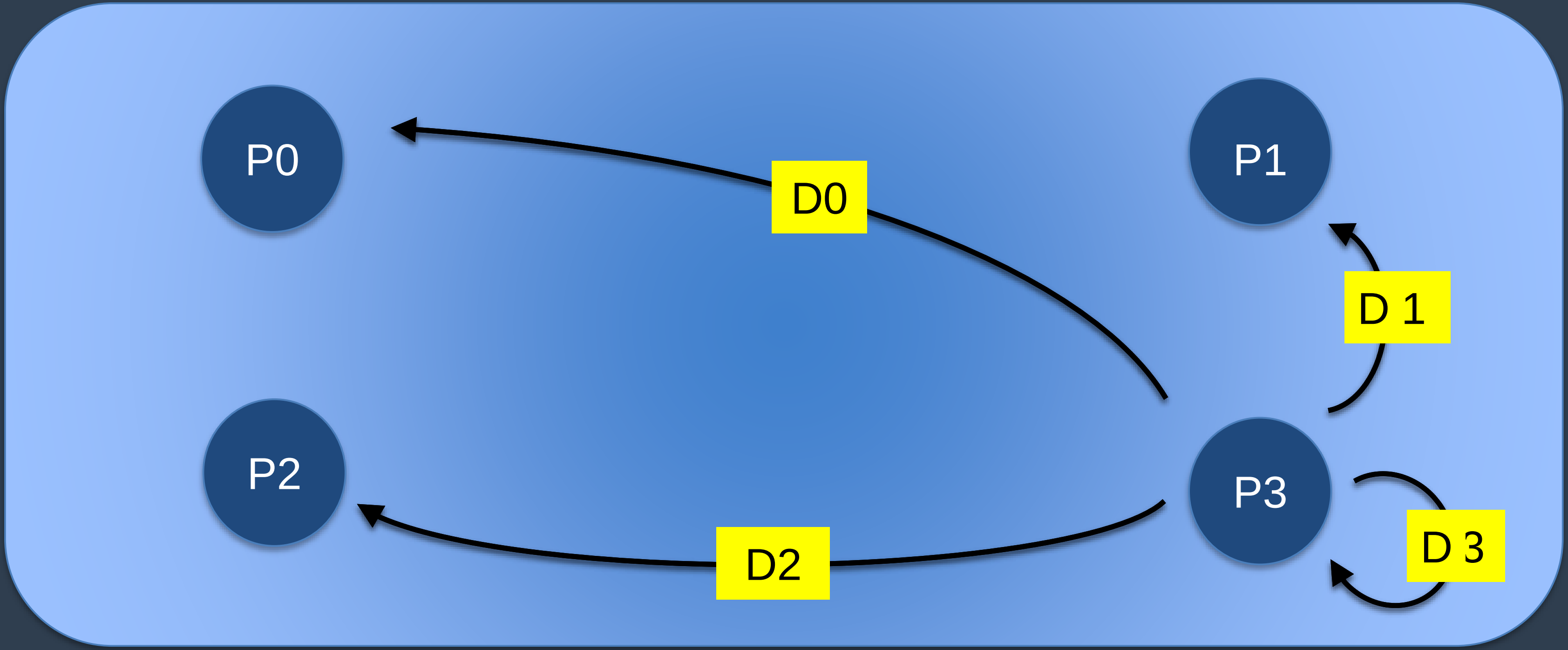


P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

$\xrightarrow{\text{MPI\_ALLTOALL}}$

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

MPI_ALLTOALL



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

$\xrightarrow{\text{MPI\_ALLTOALL}}$

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

MPI_ALLTOALL

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

Argument	Description	Intent
SBUFF	The array being sent	Input
SCOUNT	Number of items being sent	Input
SEND_TYPE	Type of SBUFF (eg. MPI_REAL)	Input
RBUFF	The array being received	Output
RCOUNT	The number of elements to receive	Input
RECEIVE_TYPE	Type of SBUFF (eg. MPI_REAL)	Input

Reduction routines

- Perform both communications and simple maths
  - sum, min, max, ... over a communicator group
- 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`

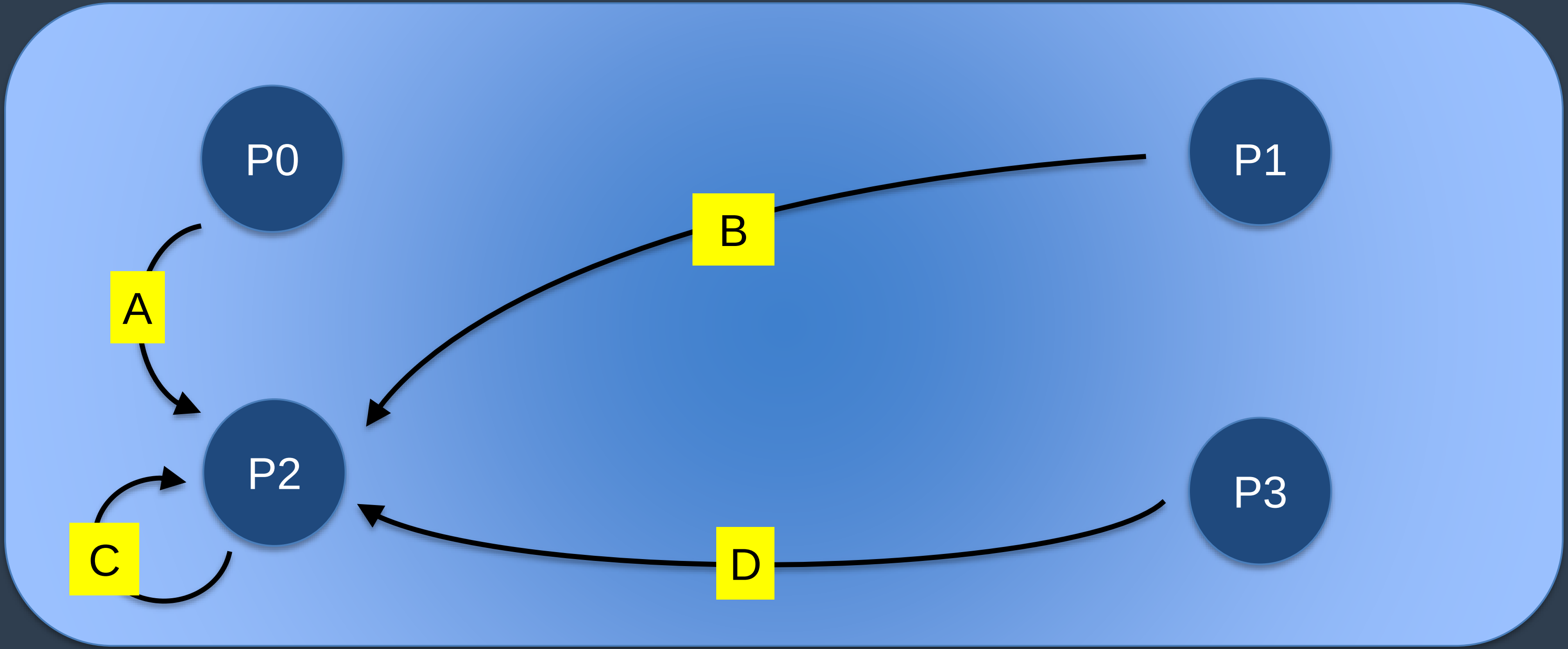
MPI_REDUCE ("sum")

P0	A			
P1	B			
P2	C			
P3	D			

MPI_REDUCE →

P0				
P1				
P2				
P3				

MPI_REDUCE ("sum")

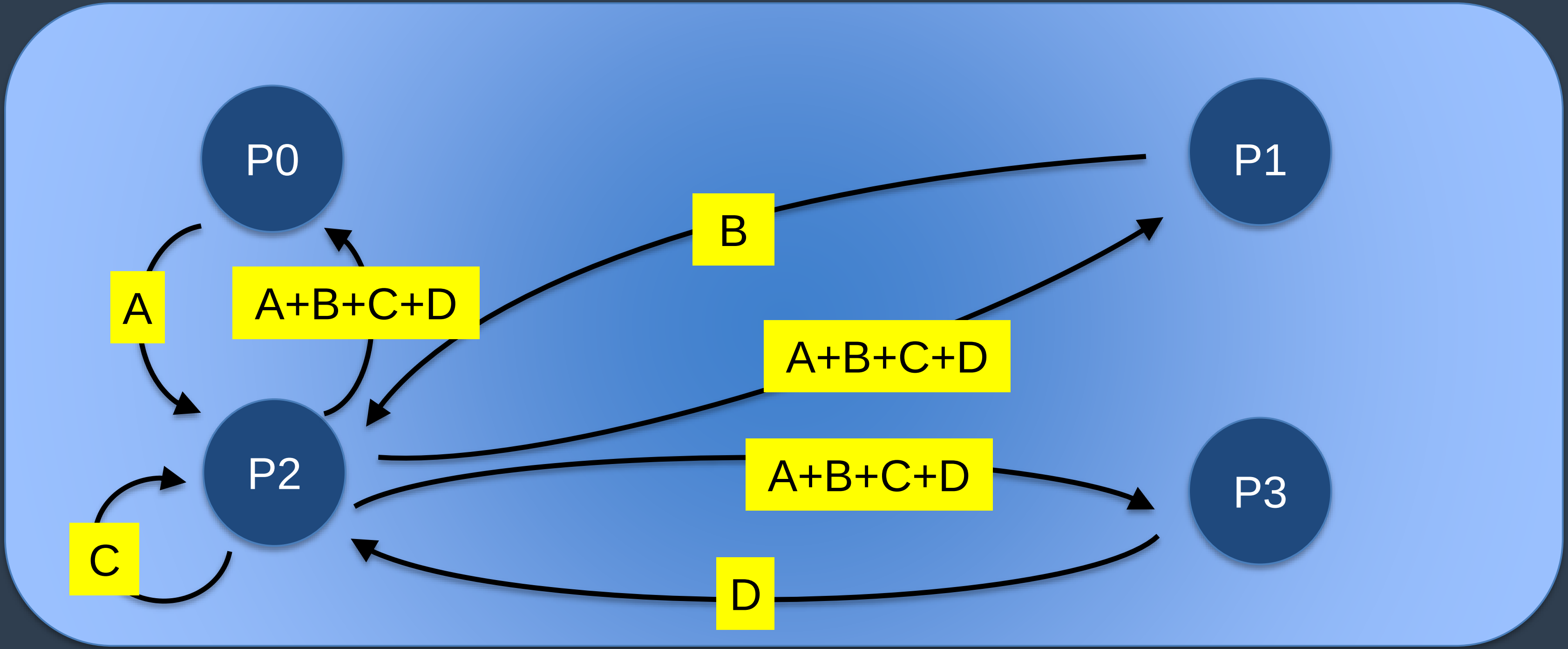


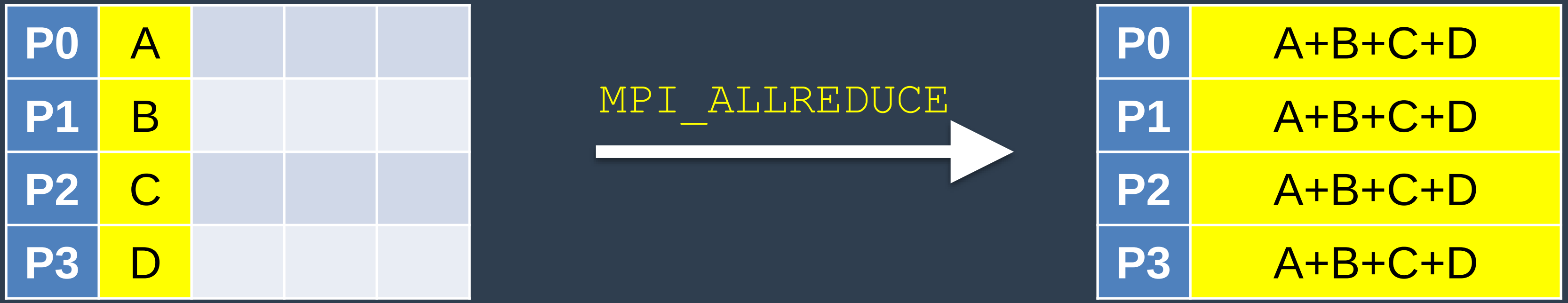
P0	A			
P1	B			
P2	C			
P3	D			

$\xrightarrow{\texttt{MPI\_REDUCE}}$

P0	
P1	
P2	A+B+C+D
P3	

MPI_ALLREDUCE ("sum")





P0	A			
P1	B			
P2	C			
P3	D			

MPI_ALLREDUCE

P0	A+B+C+D
P1	A+B+C+D
P2	A+B+C+D
P3	A+B+C+D

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)
```

Argument	Description	Intent
SBUFF	The array to be reduced	Input
RBUFF	The result of the reduction	Output
COUNT	Number of items to be reduced	Input
MPI_TYPE	Type of SBUFF (eg. MPI_REAL)	Input
OP_TYPE	Describe the reduction operation required MPI_MAX, MPI_MIN, MPI_SUM, MPI_IPROD, MPI_IAND, MPI_BAND, MPI_IOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC	Input

Back to “simple” `MPI_SEND` & `MPI_RECV`

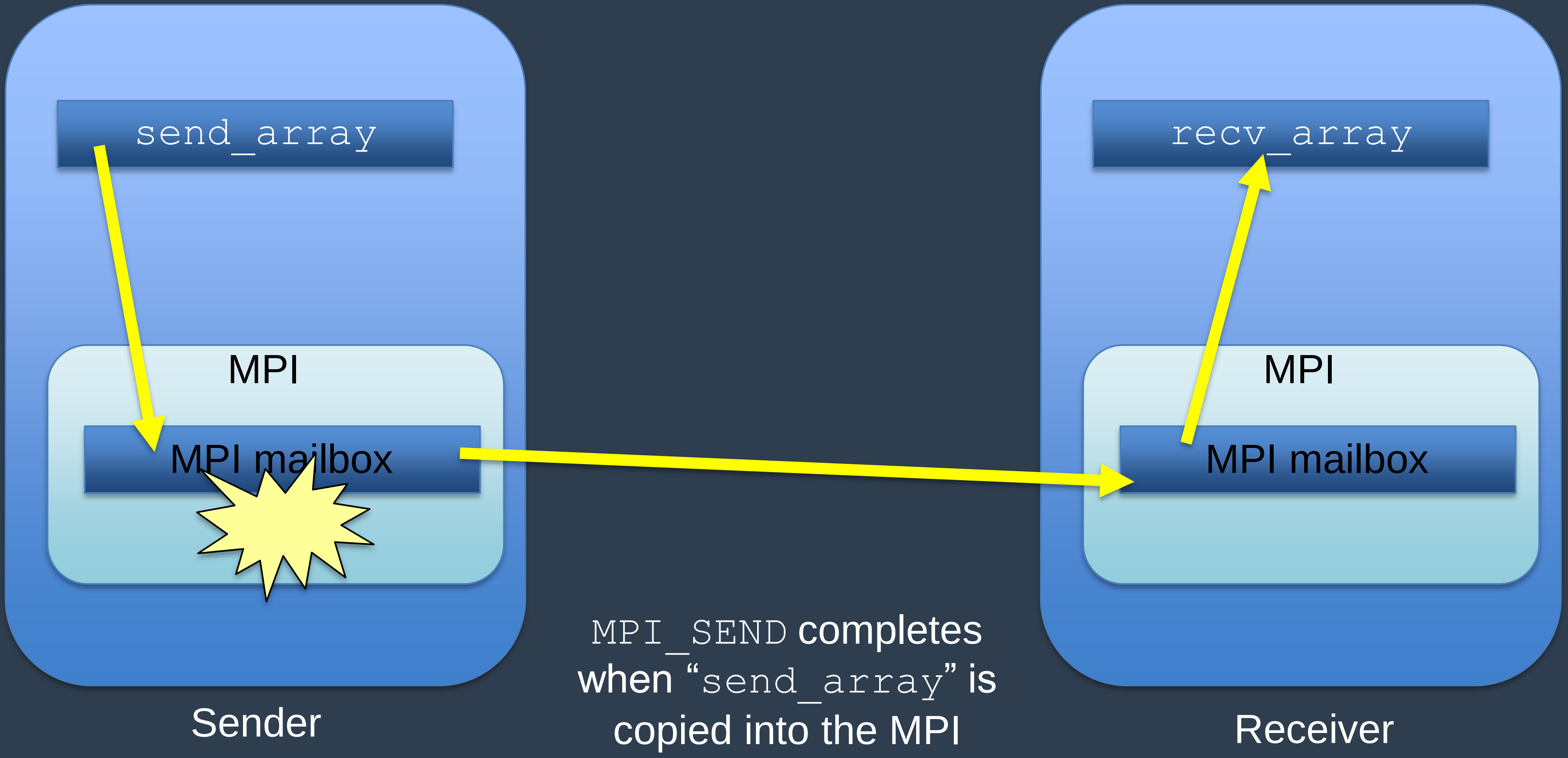
- What happens after you do MPI_SEND?
  - When does the next instruction get executed?
- What happens after you do MPI_RECV?
  - When does the next instruction get executed?
- Answer:
  - It depends!

Blocking vs Non-blocking Communications

- Blocking communication
  - Call to MPI “sending” routine does not return until the “send” buffer (array) is safe to use again
    - This does not necessarily mean the data has been sent and received by the remote task (although it might!)
  - Call to MPI “receiving” routine does not return until the “receive” buffer has received all the data in the incoming message
- Non-blocking communication
  - Call to MPI routine returns immediately
  - Further MPI calls are required to check the progress of the communication
  - Allows other work to be done during communication
- Cray’s `MPI_SEND` can sometimes be blocking and sometimes non-blocking!
  - The MPI standard doesn’t mandate whether `MPI_SEND` should be blocking or not
  - Two different behaviours, dependent on the message length…

`MPI_SEND` : Eager protocol

`MPI_SEND(send_array)`

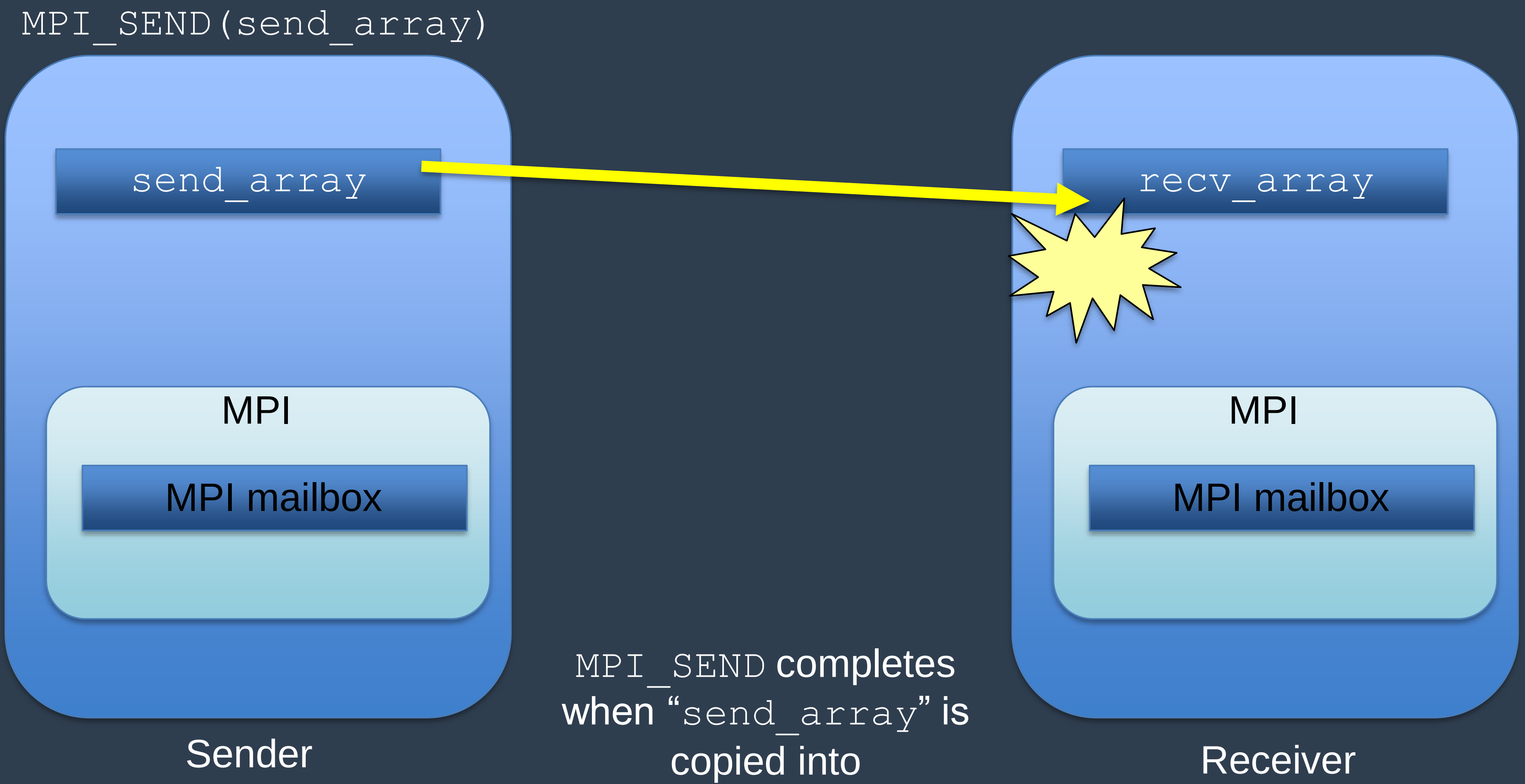


`MPI_SEND` completes when “`send_array`” is copied into the MPI mailbox on the sending task

`MPI_SEND` : Eager Protocol

- The MPI layer has copied the data elsewhere
  - using internal buffer/mailbox space on the sending task
- `MPI_SEND` returns as soon as the message has been copied
  - The message is then “in transit” but not necessarily in the receivers array
- Used for short messages
  - By default “short” is 8192 bytes (8Kb) on the Cray
  - Can be modified by environment variable
    - `$ export MPICH_GNI_MAX_EAGER_MSG_SIZE=X` *(bytes)*
    - Maximum permitted value 131072 bytes (128Kb)
- No need to worry if the remote task has done an “`MPI_RECEIVE`”
  - This is a non-blocking protocol

`MPI_SEND` : Rendezvous protocol



`MPI_SEND` completes when "`send_array`" is copied into "`recv_array`" on the receiving task

MPI_SEND : Rendezvous Protocol

- MPI_SEND does not return until the message has been successfully received by the remote task
- Used for long messages
  - By default “long” is >8192 bytes on the Cray
- Need to ensure that remote task is doing an “MPI_RECEIVE” otherwise we may deadlock…
  - Easily done!
  - eg. ping-pong example – 2 tasks exchanging messages…

```
if(me .eq.0) then
  other=1
else
  other=0
endif

call MPI_SEND(sbuff,n,MPI_REAL8,other,tag,MPI_COMM_WORLD,ierror)
call MPI_RECV(rbuff,n,MPI_REAL8,other,tag,MPI_COMM_WORLD,stat,ierror)
```

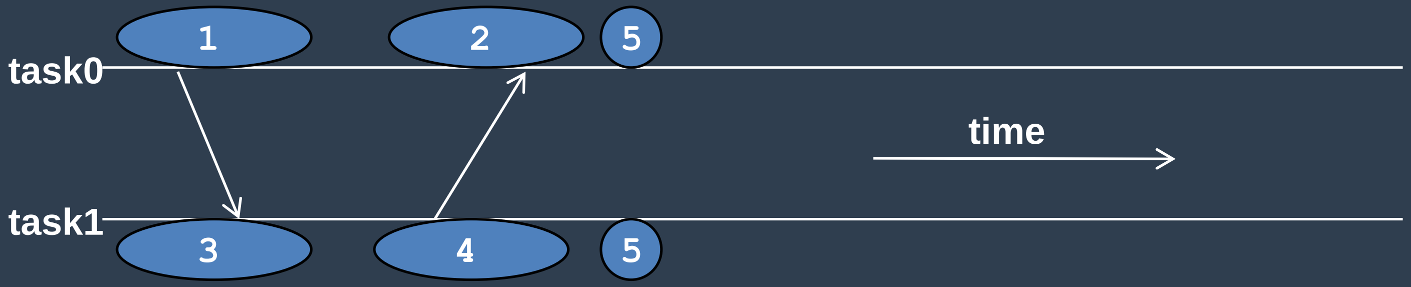
Solutions to Send/Send deadlocks

- Best advice – avoid `MPI_SEND/MPI_RECV`!
  - Behaviour is implementation dependent – code may work, but then stop working when message size changes or move to another platform
- Pair up sends and receives (next slide shows how…)
  - But this is not very efficient
- Use `MPI_SENDRECV`
  - Hopefully more efficient
- Use a buffered send (like the eager protocol, but user space buffering)
  - `MPI_BSEND`
- Use asynchronous sends/receives (recommended)
  - `MPI_ISEND` or `MPI_IRECV`

Paired Sends and Receives

- More complex code, and close synchronisation
- Less efficient
  - task 1 has to wait until it has received message from task 0 before it can send its message

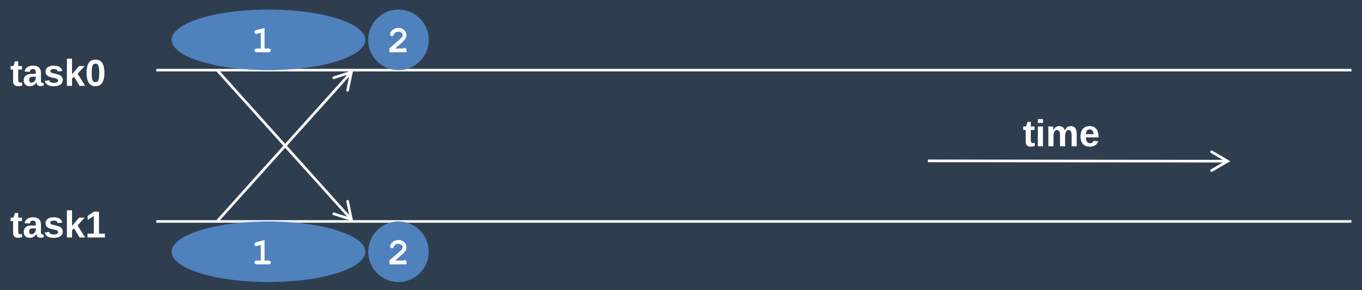
```
 if (me .eq. 0) then
   other=1
1  call MPI_SEND(sbuff,n,MPI_REAL8,other,tag,MPI_COMM_WORLD,ierror)
2  call MPI_RECV(rbuff,n,MPI_REAL8,other,tag,MPI_COMM_WORLD,stat,ierror)
 else
   other=0
3  call MPI_RECV(rbuff,n,MPI_REAL8,other,tag,MPI_COMM_WORLD,stat,ierror)
4  call MPI_SEND(sbuff,n,MPI_REAL8,other,tag,MPI_COMM_WORLD,ierror)
 endif
5
```



MPI_SENDRECV

- Simpler to code & hopefully more efficient
- Still implies close synchronisation

```
1   call MPI_SENDRECV(sbuff,n,MPI_REAL8,other,1,  &
                      rbuff,n,MPI_REAL8,other,1,  &
                      MPI_COMM_WORLD,stat,ierror)
2
```



`MPI_BSEND`

- This performs a send using an additional buffer
  - the buffer is allocated by the program via `MPI_BUFFER_ATTACH`
  - done once as part of the program initialisation
  - `MPI_BSEND` completes as soon as message is copied into buffer
- Typically quick to implement
  - add the `MPI_BUFFER_ATTACH` call
    - how big to make the buffer?
  - change `MPI_SEND` to `MPI_BSEND` everywhere
- But introduces additional memory copy
  - extra overhead
  - not recommended for production codes
  - One day your buffer won’t be big enough!

MPI_IRECV & MPI_ISEND

- Uses Non Blocking Communications
- "I" stands for immediate
  - the call returns immediately
- Routines return without completing the operation
  - the operations run asynchronously (in the background)
  - Must NOT reuse the buffer (send/receive array) until safe to do so
- Later test that the operation completed
  - via an integer identification handle "request" passed to `MPI_WAIT`

```
call MPI_IRECV(rbuff,n,MPI_REAL8,other,1,MPI_COMM_WORLD,request,ierror)
call MPI_SEND (sbuff,n,MPI_REAL8,other,1,MPI_COMM_WORLD,ierror)
call MPI_WAIT(request,stat,ierr)
```

- Alternatively could have used `MPI_ISEND` and `MPI_RECV`

Non blocking communications

- Routines include
  - `MPI_ISEND`
  - `MPI_IRECV`
  - `MPI_WAIT`
  - `MPI_WAITALL`
    - Waits for a number of outstanding communications to complete
  - And many, many others!
    - See the documentation

Final Practical

- exercise2

- A “simple” numerical model

- See the README for details

- Use the links to external documentation for details of the arguments required for various MPI routines you might want to use

- Ask if you need help or don’t understand anything!