Programming with MPI

Datatypes and Collectives

Nick Maclaren

nmm1@cam.ac.uk

May 2008

Transfer Procedures

These need to specify one or more transfer buffers
Used to send or receive data, or both

These are specified using three arguments:
 The address of the buffer
 The size of the buffer
 The base datatype of the buffer

They also need to specify some control information
 The root process for 1:all transfers
 The communicator to be used for the collective

Transfer Buffers (1)

MPI transfers use vectors (i.e. 1–D arrays)
The base element datatypes are always scalars

They all include an element count argument
i.e. the length of the vector in elements

- The arguments are type–generic (choice)

Declared as “void *” in C

Fortran relies on no checking (see later)

- The datatype is passed as a separate argument

Transfer Buffers (2)

The vectors are always contiguous arrays
Each element immediately follows its predecessor

Like Fortran 77 or C arrays, not all of Fortran 90
Return to Fortran 90 assumed shape arrays later

For example, consider transferring 100 integers
The element count is 100

These are declared like:
Fortran: INTEGER BUFFER (100)
C: int buffer [100] ;

C++ Classes

But what about C++ library containers?
In general, they are not contiguous arrays

- But <vector>, <array> and <string> are
  ⇒ Though not <vector<bool> >!

In all cases, & object . front ( ) is the data address
So is & object [0], and many people use that instead
Conversion to void * in the call is automatic

- This applies far more generally than just to MPI

Also, similar remarks apply to libraries like Boost

Datatypes (1)

Datatypes are MPI constants, not language types
There is a fairly complete set that are built–in

- Note that does NOT mean language constants

Each datatype has an associated size

- Count and offsets are in units of that

Exactly as in Fortran or C arrays

```
double buffer [ 100 ] ;
MPI_Bcast ( buffer , 100 , MPI_DOUBLE ,
    root , MPI_COMM_WORLD )
```

Datatypes (2)

The MPI and language datatypes must match
Some exceptions, but I suggest avoiding them

- You will not get warned if you make an error

As in K&R C, C casts and Fortran 77
There is no C++ or Fortran 90 type–checking

In theory, a compiler could detect a mismatch
But it would have to be "MPI aware" and none are

Datatypes (3)

Here is a sample of recommended datatypes
All that you need for the first examples
We will come back to these in more detail later

Fortran:

MPI_INTEGER

MPI_DOUBLE_PRECISION

C:

MPI_INT

MPI_DOUBLE

Collectives (1)

We have already used MPI_Barrier
All of the others involve some data transfer

- All processes in a communicator are involved

For use on a subset, create another communicator
We shall come back to that later

- All datatypes and counts must be the same

    A few, obscure exceptions – not recommended

Obviously the communicator must be, too

Collectives (2)

- All of the buffer addresses may be different

MPI processes don't share any addressing

This generalises in more advanced use
The data layout may be different – see later

- Match the communicator, datatypes and counts

And call all of the collectives "at the same time"

- Easiest to achieve using the SPMD model

You can code just one collective call

Collectives (3)

Some collectives are asymmetric (1:all)
E.g. broadcast from one proc. to all communicator
That means all processes – including itself

Those all have a root process argument
This also must be the same on all processes
Any process can be specified – not just zero

Symmetric ones don't have that argument
For example, MPI_Barrier doesn't

Collectives (4)

- Most use separate send and receive buffers

Both for flexibility and for standards conformance

- Usually specify the datatype and count for each

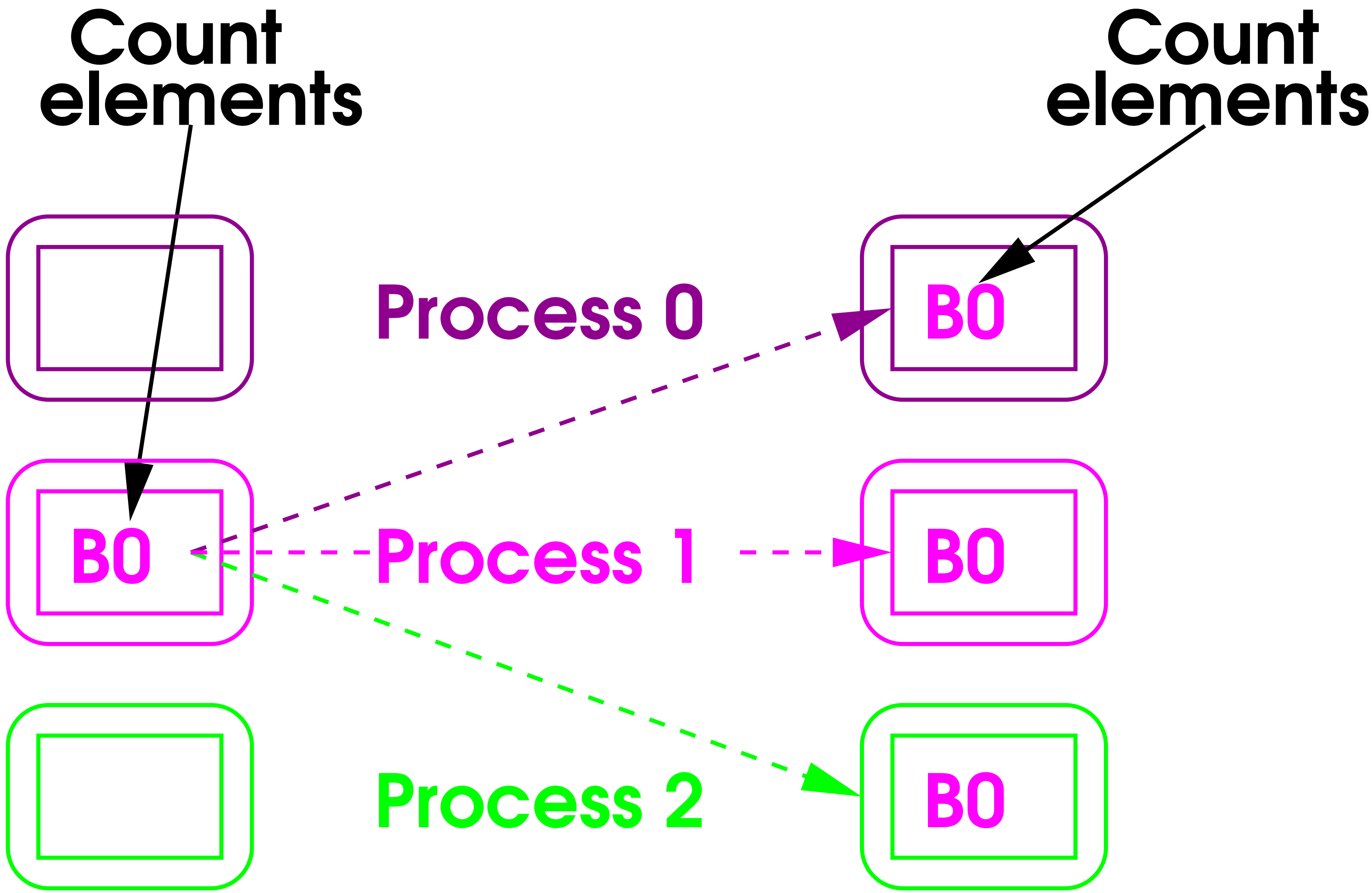
Needed for advanced features not covered here

MPI uses only the arguments it needs
I.e. unused ones are completely ignored

- Set them all compatibly – it is much safer!

Keep all datatypes and counts the same

Broadcast



Count
elements
Count
elements
Process 0
B0
B0
Process 1
B0
Process 2
B0

Broadcast (1)

Broadcast copies the same data from the root to all processes in the communicator

Fortran example:

```
REAL(KIND=KIND(0.0D0)) :: buffer ( 100 )
INTEGER , PARAMETER :: root = 3
INTEGER :: error
CALL MPI_Bcast ( buffer , 100 ,          &
     MPI_DOUBLE_PRECISION , root ,       &
     MPI_COMM_WORLD , error )
```

Broadcast (2)

C example:

```
double buffer [ 100 ] ;
int root = 3 , error ;
error = MPI_Bcast ( buffer , 100 , MPI_DOUBLE ,
        root , MPI_COMM_WORLD ) ;
```

C++ using C interface example:

```
vector<double> buffer ( 100 ) ;
int root = 3 ;
error = MPI_Bcast ( & buffer . front ( ) , 100 ,
        MPI_DOUBLE , root , MPI_COMM_WORLD ) ;
```

Multiple Transfer Buffers

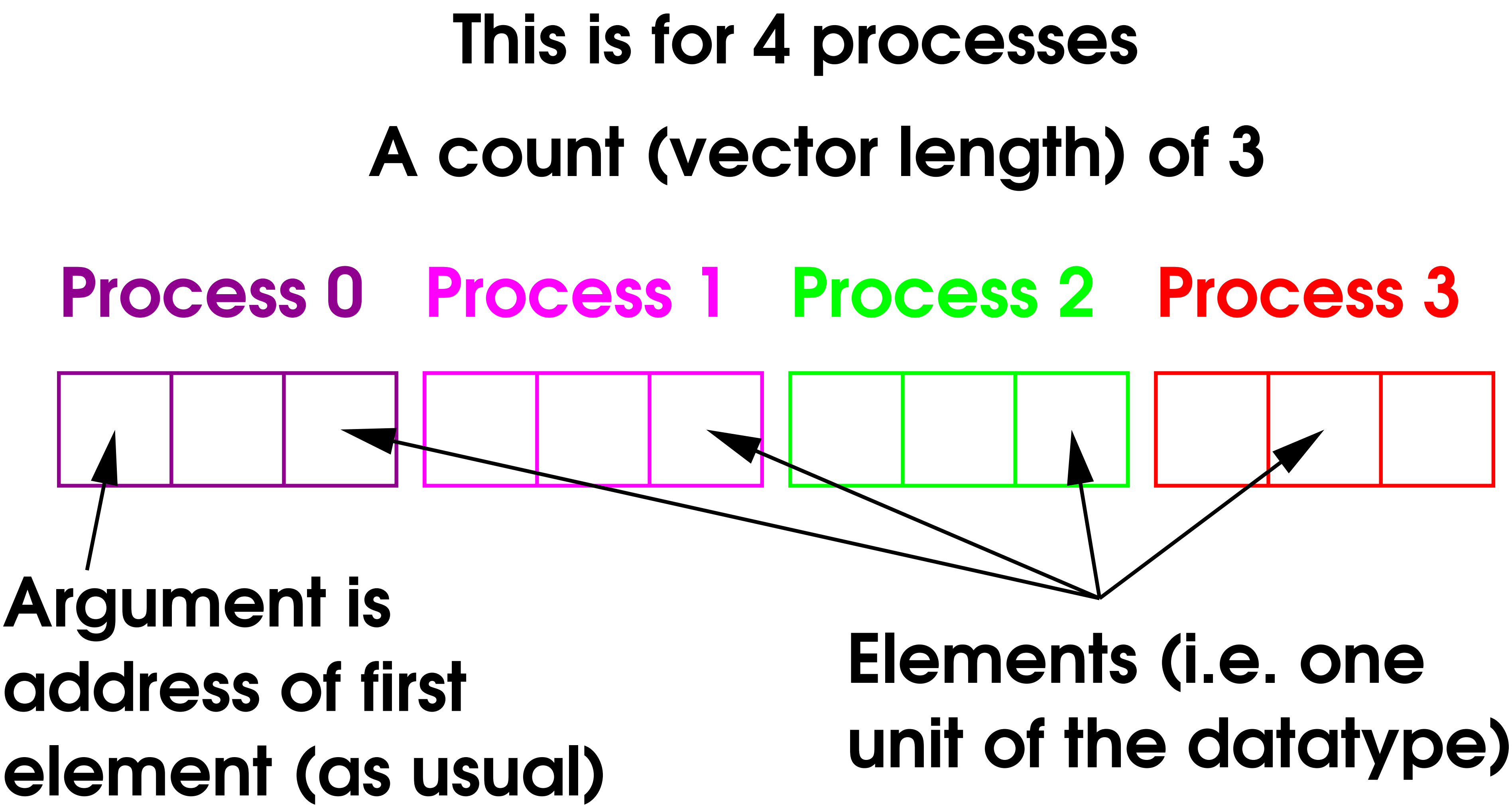
Many collectives need one buffer per process

For example, take a 1⇒N scatter operation
The root sends different data to each process

Each pairwise transfer buffer is concatenated
in the order of process numbers (i.e. 0...N–1)

Size of source = N * size of each result

Multiple Transfer Buffers



Size Specifications (1)

Size specifications are slightly counter–intuitive
That is done for consistency and simplicity

You specify the size of each pairwise transfer
MPI will deduce the total size of the buffers
I.e. it will multiply by process count, if needed

- The process count is implicit
It is taken from the communicator
I.e. the result from MPI_Comm_size

Size Specifications (2)

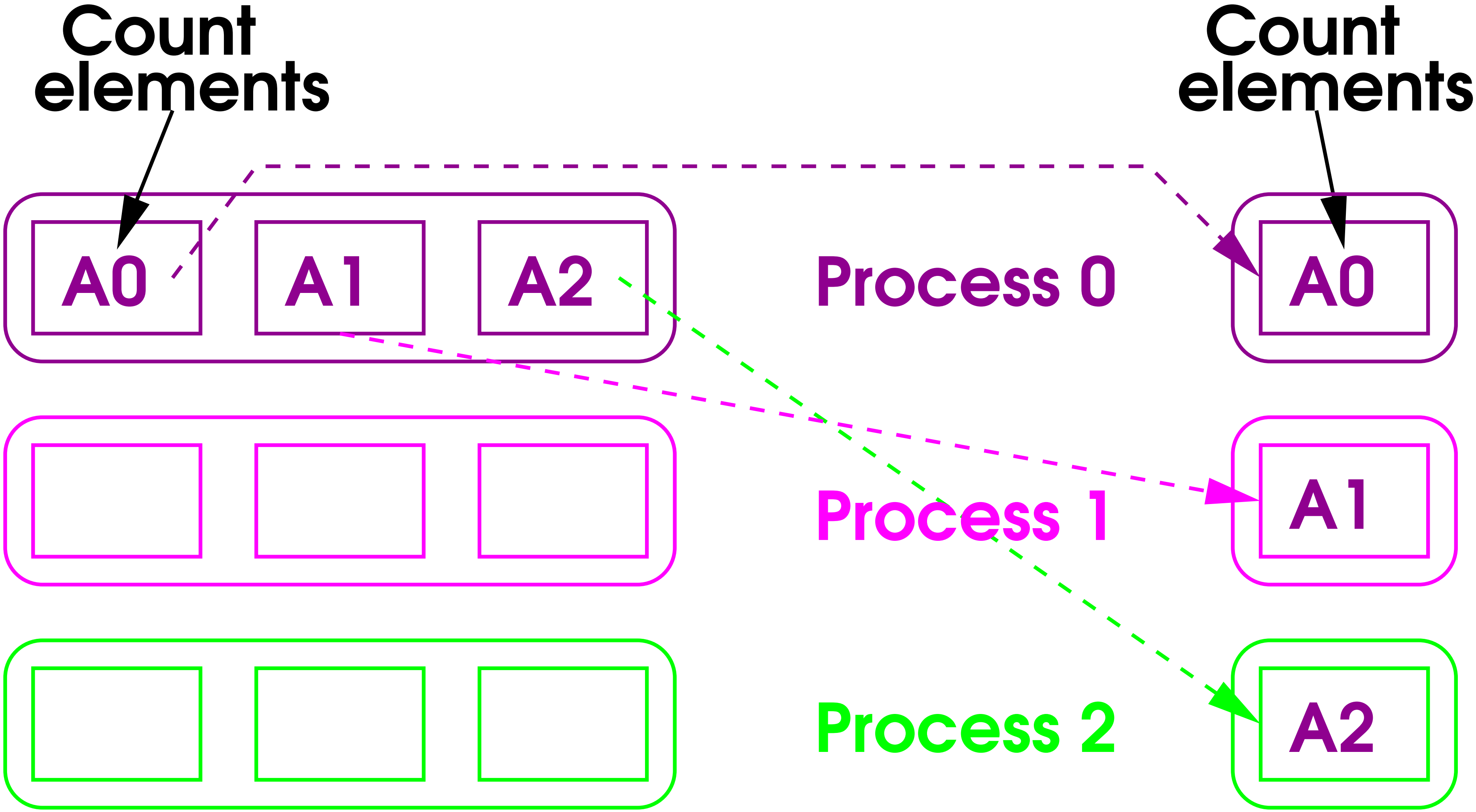
“void *” defines no length in C
Nor does “<type> :: buffer(*)” in Fortran

- It is up to you to get it right

No compiler can trap an error with that

We shall use scatter as our first example
This is one process sending different data
to every process in the communicator

Scatter


Count
elements
A0
A1
A2
Process 0
Count
elements
A0
Process 1
A1
Process 2
A2

Scatter (1)

Scatter copies different data from the root
to all processes in the communicator

The send buffer is used only on the root
The receive buffer is used on all processes

Following examples assume $\leq$30 processes
Specified only in the send buffer size

- Note the differences in the buffer declarations

Scatter (2)

Fortran example:

```
REAL(KIND=KIND(0.0D0)) ::          &
        sendbuf ( 100 , 30 ) , recvbuf ( 100 )
INTEGER , PARAMETER :: root = 3
INTEGER :: error
CALL MPI_Scatter (          &
    sendbuf , 100 , MPI_DOUBLE_PRECISION ,       &
    recvbuf , 100 , MPI_DOUBLE_PRECISION ,       &
        root , MPI_COMM_WORLD , error )
```

Scatter (3)

C example:

```
double sendbuf [ 30 ] [ 100 ] , recvbuf [ 100 ] ;
int root = 3 , error ;
error = MPI_Scatter (
        sendbuf , 100 , MPI_DOUBLE ,
        recvbuf , 100 , MPI_DOUBLE ,
        root , MPI_COMM_WORLD )
```

Scatter (4)

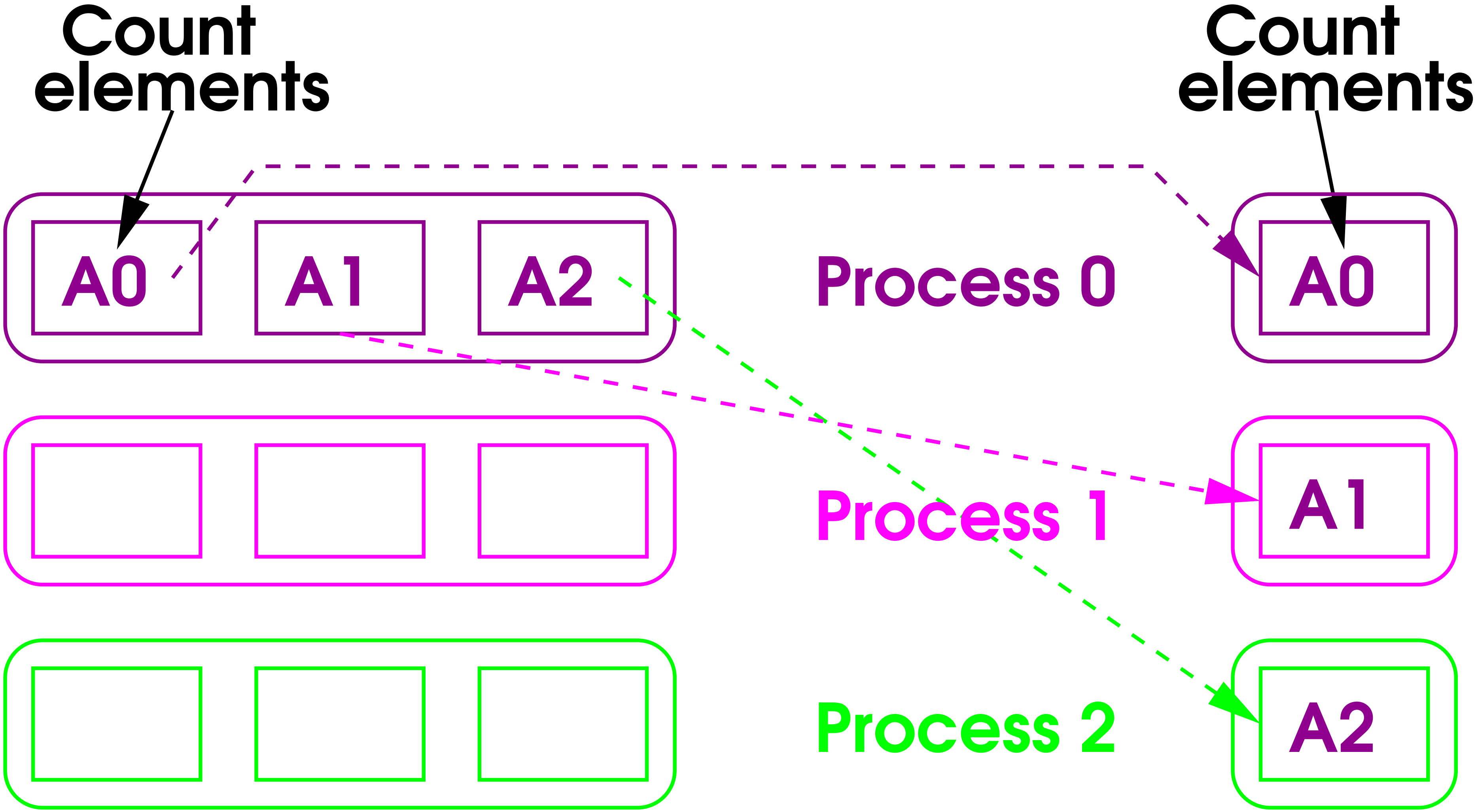
C++ using C interface example:

```
vector < double > sendbuf ( 30 * 100 ) , recvbuf ( 100 ) ;
int root = 3 ;
error = MPI_Scatter (
    & sendbuf . front ( ) , 100 , MPI_DOUBLE ,
    & recvbuf . front ( ) , 100 , MPI_DOUBLE ,
    root , MPI_COMM_WORLD )
```

Remember that only the contents are contiguous
Do NOT create multiple buffers like this:

```
array< array< double , 100 > , 30 > sendbuf ;
```

Scatter



Count
elements
A0
A1
A2
Process 0
A0
Count
elements
Process 1
A1
Process 2
A2

Hiatus

That is the basic principles of collectives

Now might be a good time to do some examples
The first few questions cover the material so far

After that, we cover datatypes more thoroughly
And describe more of the collectives

Fortran Datatypes (1)

Recommended datatypes:

MPI_CHARACTER ($\equiv$ CHARACTER(LEN=1))
MPI_LOGICAL
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX

MPI_DOUBLE_COMPLEX

I.e. COMPLEX(KIND=KIND(0.0D0))

Fortran Datatypes (2)

Fortran 90 parameterized types are also supported
REAL(KIND=SELECTED_REAL_KIND(15,300))
Details of that mechanism is given later

For use from Fortran, that's all I recommend
There are some more built–in datatypes, though

MPI_PACKED, for MPI derived datatypes

MPI_BYTE (uninterpreted 8–bit bytes)
What you can do with these is a bit restricted

Other Fortran Datatypes

And you should definitely avoid these

MPI_INTEGER1	MPI_REAL2
MPI_INTEGER2	MPI_REAL4
MPI_INTEGER4	MPI_REAL8

MPI_<type>N translates to <type>*N

That notation is non–standard and outmoded

- It doesn't mean the size in bytes!

E.g. REAL*2 works only on Cray vector systems

C Datatypes (1)

MPI_CHAR is for char, meaning characters

Don't use it for small integers and arithmetic

Recommended integer datatypes:

- MPI_UNSIGNED_CHAR
- MPI_SIGNED_CHAR
- MPI_SHORT
- MPI_UNSIGNED_SHORT
- MPI_INT
- MPI_UNSIGNED (not MPI_UNSIGNED_INT)
- MPI_LONG
- MPI_UNSIGNED_LONG

C Datatypes (2)

Recommended floating–point datatypes:

- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE

For use from C, I recommend one more

MPI_BYTE (uninterpreted 8–bit bytes)
What you can do with these is a bit restricted

C++ Datatypes

Warning: these all depend on MPI 3.0
Most implementations won't have them yet

MPI_CXX_BOOL
MPI_CXX_FLOAT_COMPLEX
MPI_CXX_DOUBLE_COMPLEX
MPI_CXX_LONG_DOUBLE_COMPLEX

They all correspond to the obvious C++ type

C99 and Complex

There are types for C99 _Complex, if you use it
But I don't advise using that (or most of C99)

C99 _Complex may not be compatible with C++
And WG14 have now made _Complex optional

MPI_C_FLOAT_COMPLEX
MPI_C_COMPLEX is a synonym
MPI_C_DOUBLE_COMPLEX
MPI_C_LONG_DOUBLE_COMPLEX

Other C Datatypes

I don’t recommend the other built–in datatypes

MPI_LONG_LONG_INT (note the name)

Needs C99 and optional, anyway

MPI_UNSIGNED_LONG_LONG

It need C99 and is optional, anyway

MPI_WCHAR (whatever C wchar_t is)

No useful specification in C90, C99 or C++

MPI_PACKED, for MPI derived datatypes

There is no support for C99’s new integer types

- Ask me offline why that is a Good Thing

C++, C99 and Complex

Some types for C++, including bool and complex

And quite a lot of types for C99 extensions

Omitted, for simplicity, because there are problems
The handouts describe the situation in more detail

Gather

Gather is precisely the converse of scatter

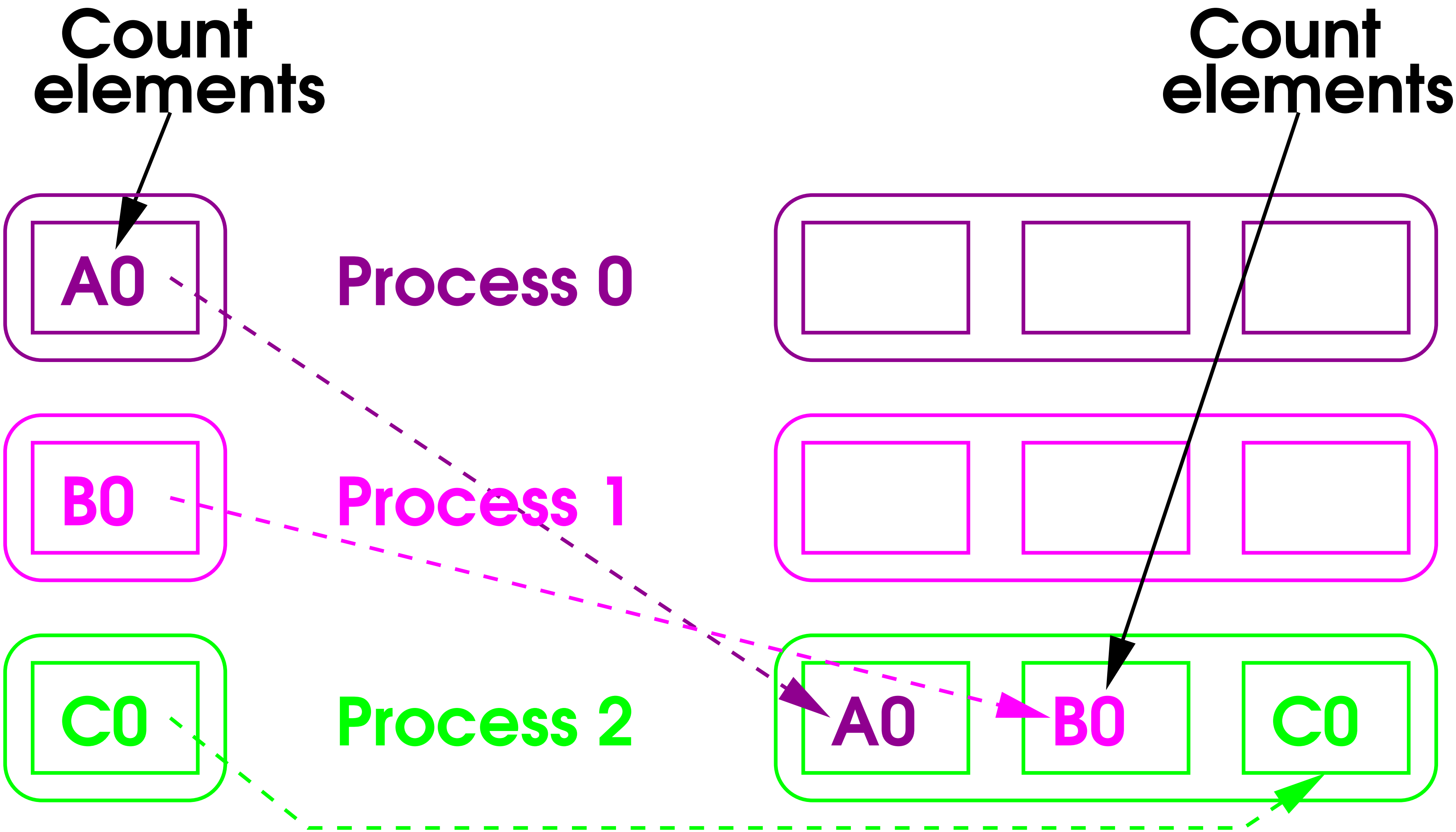
- Just change MPI_Scatter to MPI_Gather

Of course, the array sizes need changing

- It is the receive buffer that needs to be bigger

The send buffer is used on all processes
The receive buffer is used only on the root

Gather


Count
elements
Count
elements
A0
Process 0
B0
Process 1
C0
Process 2
A0
B0
C0

Allgather (1)

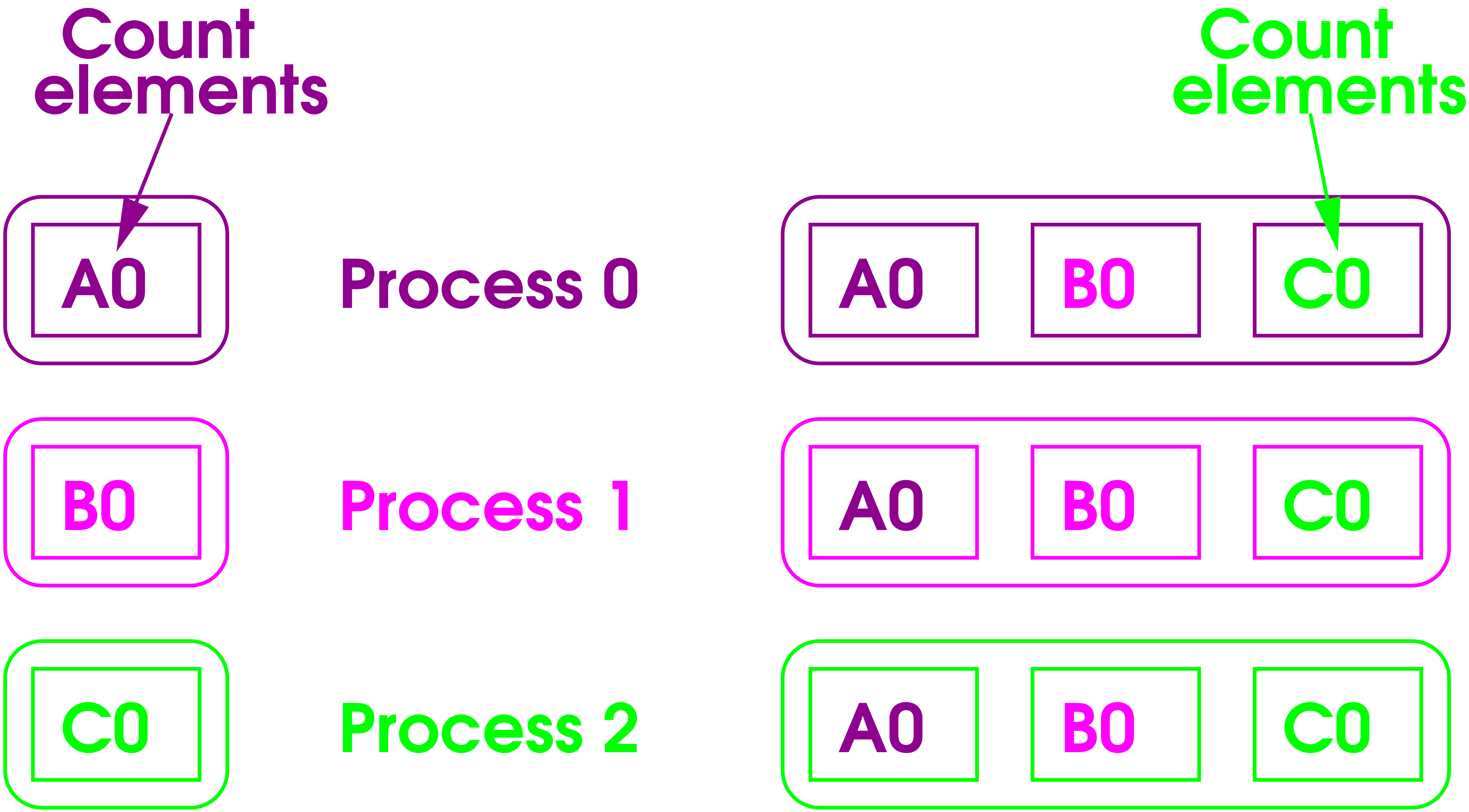
You can gather data and then broadcast it
The interface is very similar, with one difference

- This is now a symmetric operation
So has no argument specifying the root process

- Change MPI_Gather to MPI_Allgather
And remove the root process argument, of course

- The receive buffer is now used on all processes

Allgather


Count elements
Count elements
A0
Process 0
A0
B0
C0
B0
Process 1
A0
B0
C0
C0
Process 2
A0
B0
C0

Allgather (2)

Fortran example:

```
REAL(KIND=KIND(0.0D0)) ::         &
      sendbuf ( 100 ) , recvbuf ( 100 , 30 )
INTEGER :: error
CALL MPI_Allgather (         &
   sendbuf , 100 , MPI_DOUBLE_PRECISION ,      &
   recvbuf , 100 , MPI_DOUBLE_PRECISION ,      &
      MPI_COMM_WORLD , error )
```

Allgather (3)

C example:

```
double sendbuf [ 100 ] , recvbuf [ 30 ] [ 100 ] ;
int error ;
error = MPI_Allgather (
        sendbuf , 100 , MPI_DOUBLE ,
        recvbuf , 100 , MPI_DOUBLE ,
        MPI_COMM_WORLD )
```

Alltoall

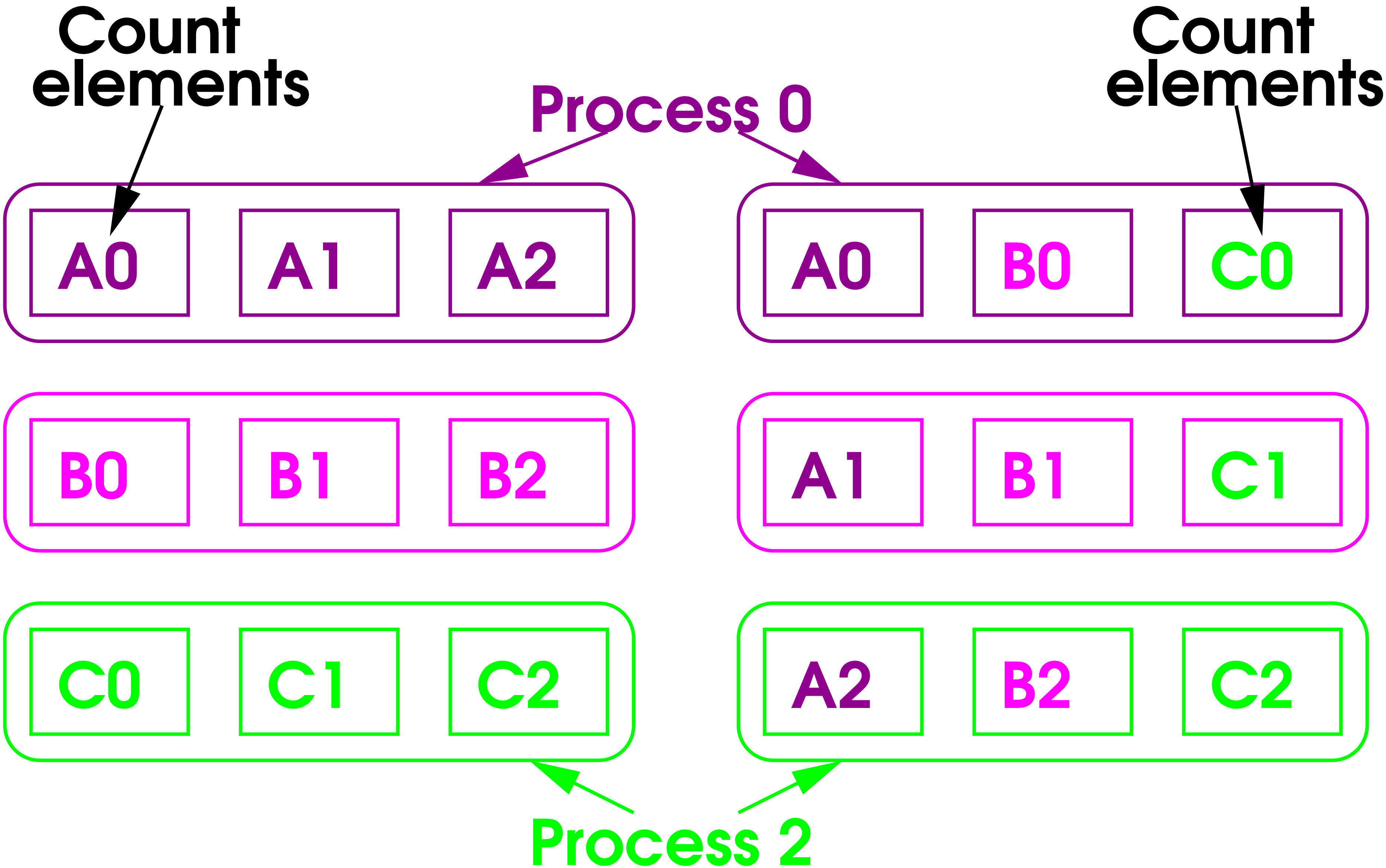
You can do a composite gather/scatter operation
Essentially the same interface as MPI_Allgather

- Just change MPI_Allgather to MPI_Alltoall
- Now, both buffers need to be bigger

Think of this as a sort of parallel transpose
Used when implementing matrix transpose

- It's very powerful – a key for performance

Alltoall


Count
elements
Process 0
Count
elements
A0
A1
A2
A0
B0
C0
B0
B1
B2
A1
B1
C1
C0
C1
C2
A2
B2
C2
Process 2

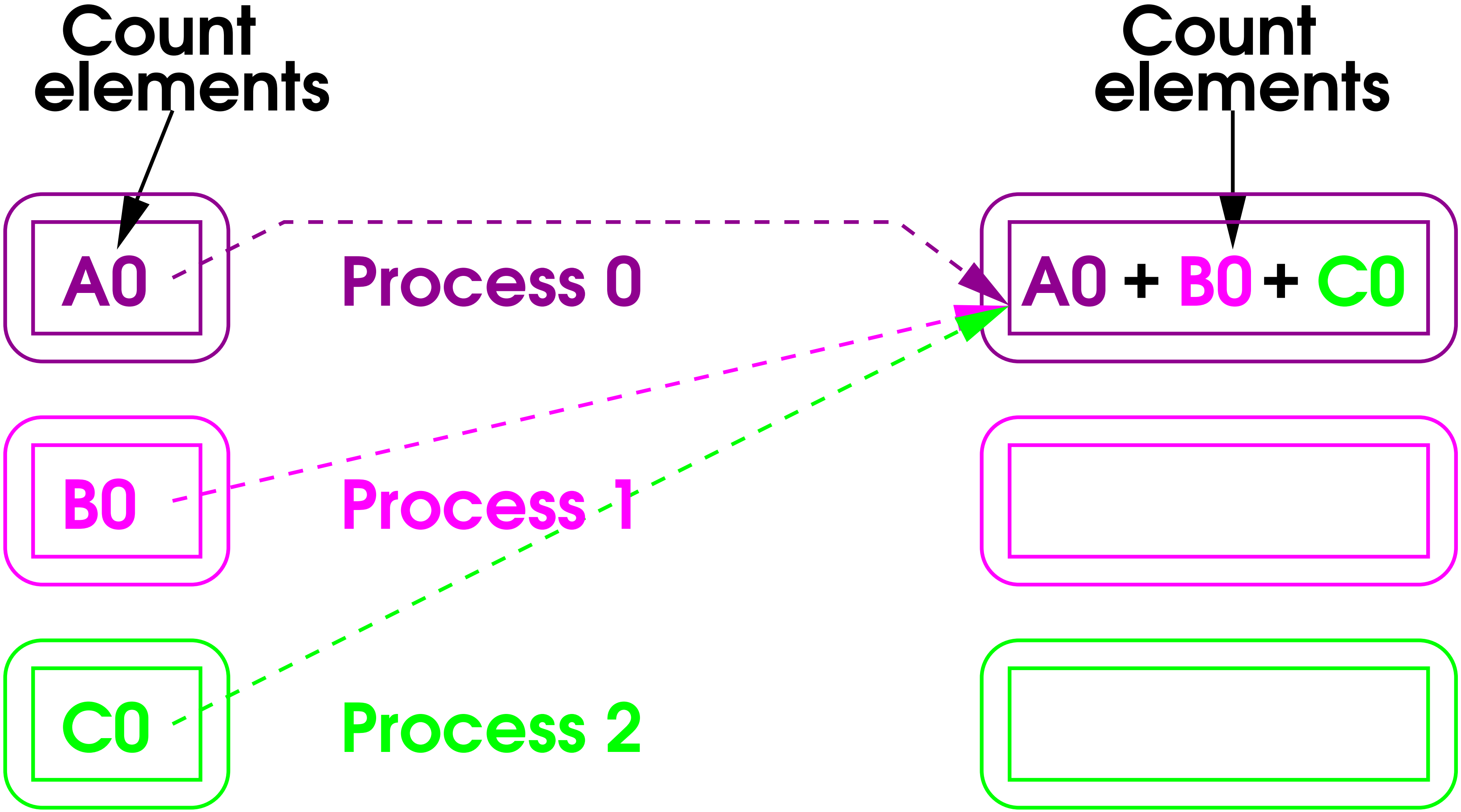
Global Reductions (1)

One of the basic parallelisation primitives

Start with a normal gather operation
Then sum the values over all processes
Often can be implemented much more efficiently

- Summation is not the only reduction
Anything that makes mathematical sense
All of the standard ones are provided

Reduce



Count
elements
Count
elements
A0
Process 0
A0 + B0 + C0
B0
Process 1
C0
Process 2

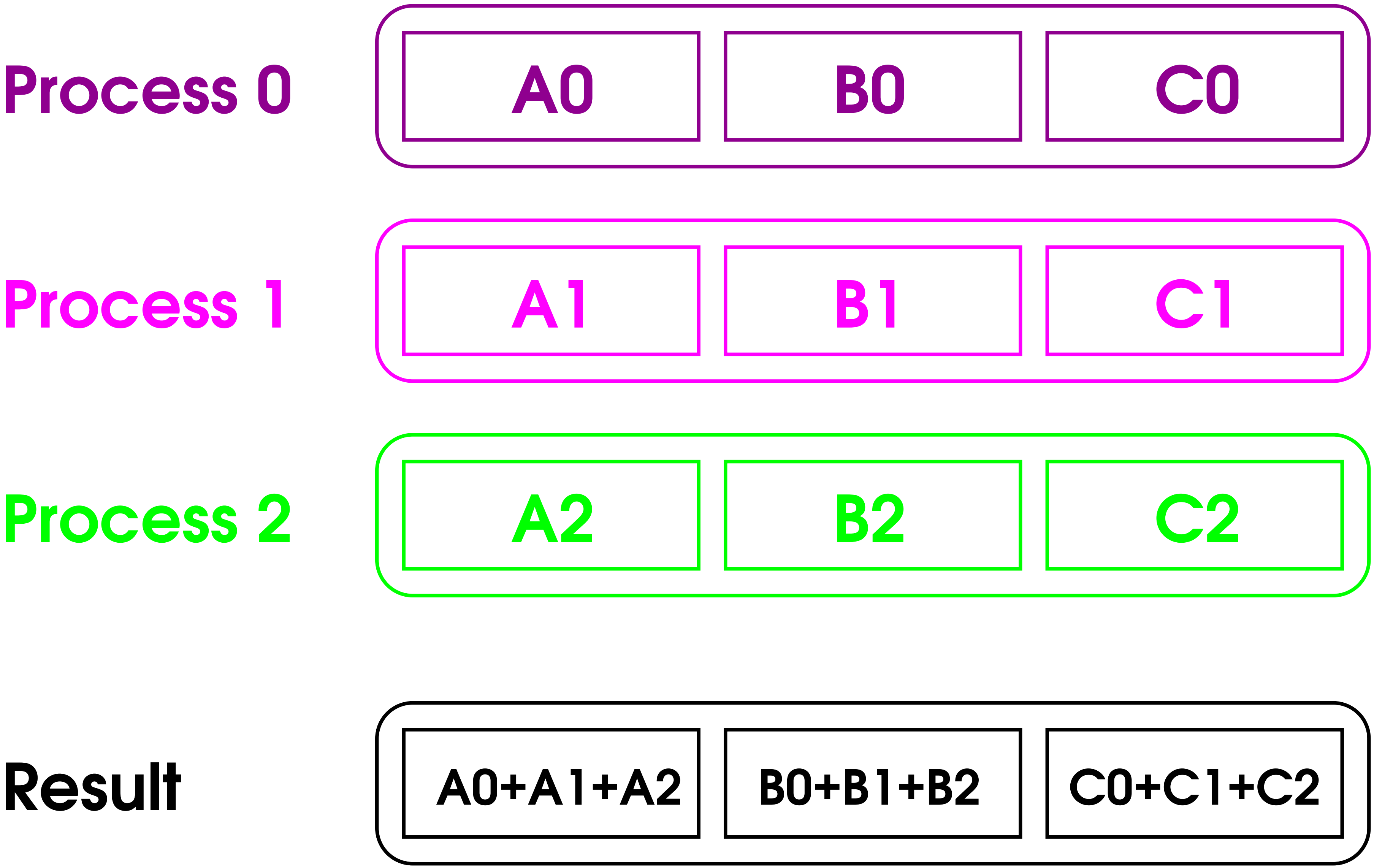
Global Reductions (2)

- It specifies the datatype and count once
Not separately for the source and result
It makes no sense to do that, so MPI doesn't

- Does not reduce over the vector
The count is the size of the result, too
It sums the values for each index separately

You have to reduce over the vector yourself
- Doing it beforehand is more efficient

Reduce Result



Reduce (2)

Fortran example:

```
REAL(KIND=KIND(0.0D0)) ::          &
        sendbuf ( 100 ) , recvbuf ( 100 )
INTEGER , PARAMETER :: root = 3
INTEGER :: error
CALL MPI_Reduce ( sendbuf , recvbuf ,        &
        100 , MPI_DOUBLE_PRECISION ,         &
        MPI_SUM , root , MPI_COMM_WORLD , error )
```

Reduce (3)

C example:

```
double sendbuf [ 100 ] , recvbuf [ 100 ] ;
int root = 3 , error ;
error = MPI_Reduce ( sendbuf , recvbuf ,
        100 , MPI_DOUBLE , MPI_SUM , root ,
        MPI_COMM_WORLD )
```

Allreduce

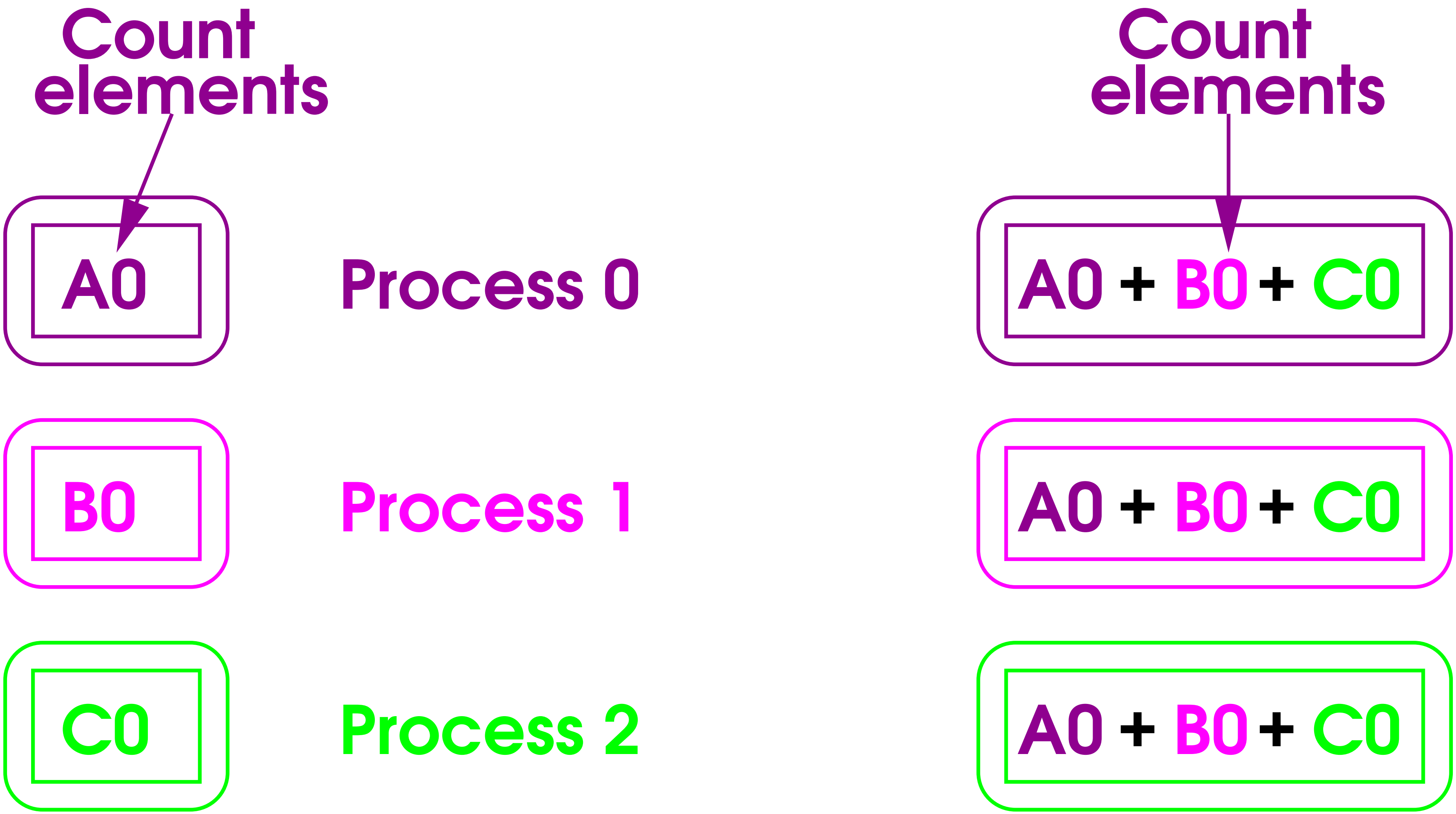
You can reduce data and then broadcast it
Again, the interface is essentially identical

- This is now a symmetric operation
So has no argument specifying the root process

- Just change MPI_Reduce to MPI_Allreduce
And remove the root process argument, of course

- The receive buffer is now used on all processes

Allreduce



Count
elements
A0
Process 0
B0
Process 1
C0
Process 2
Count
elements
A0 + B0 + C0
A0 + B0 + C0
A0 + B0 + C0

Reduction Operations (1)

Same rules for all precisions of number

MPI_MIN	integer or real minimum
MPI_MAX	integer or real maximum
MPI_SUM	integer, real or complex sum
MPI_PROD	integer, real or complex product

Note there are no reductions on character data

Reduction Operations (2)

Boolean is int in C and LOGICAL in Fortran
The supported values are only True and False

You can also perform bitwise operations on integers

MPI_LAND	Boolean AND
MPI_LOR	Boolean OR
MPI_LXOR	Boolean Exclusive OR
MPI_BAND	integer bitwise AND
MPI_BOR	integer bitwise OR
MPI_BXOR	integer bitwise Exclusive OR

More on Collectives

There is a little more to say on collectives
But that's quite enough for now

The above has covered all of the essentials
The remaining aspects to cover are:

- Fortran parameterised types
- Searching as a reduction
- Using more flexible buffer layouts

- Using collectives efficiently

Practicals

There are a lot of exercises on the above
Will take you through almost all aspects

- Each one should need very little editing/typing

You can start from a previous one as a basis

PLEASE check you understand the point
And that you get the same answers as are provided
And that you understand what it is doing and why

- They are pointless if you do them mechanically