Programming with MPI

More on Datatypes and Collectives

Nick Maclaren

nmm1@cam.ac.uk

May 2008

Less Basic Collective Use

A few important facilities we haven't covered
Less commonly used, but fairly often needed
In particular, one of them can help a lot with I/O

And then how to use collectives efficiently
Plus one potentially useful minor feature

Fortran Precisions (1)

Fortran 90 allows selectable precisions
KIND=SELECTED_INTEGER_KIND(precision)
KIND=SELECTED_REAL_KIND(precision[,range])

Can create a MPI derived datatype to match these
Then can use it just like a built–in datatype

Surprisingly, it is a predefined datatype
Do NOT commit or free it
[Don't worry if that makes no sense to you]

Fortran Precisions (2)

```
INTEGER ( KIND =        &
          SELECTED_INTEGER_KIND ( 15 ) ) ,      &
       DIMENSION ( 100 ) :: array
INTEGER :: root , integertype , error

CALL MPI_Type_create_f90_integer (       &
       15 , integertype , error )
CALL MPI_Bcast ( array , 100 ,       &
       integertype , root ,       &
       MPI_COMM_WORLD , error )
```

Fortran Precisions (3)

REAL and COMPLEX are very similar

```
REAL ( KIND =       &
        SELECTED_REAL_KIND ( 15 , 300 ) ) ,     &
     DIMENSION ( 100 ) :: array
CALL MPI_Type_create_f90_real (       &
     15 , 300 , realtype , error )

COMPLEX ( KIND =       &
        SELECTED_REAL_KIND ( 15 , 300 ) ) ,     &
     DIMENSION ( 100 ) :: array
CALL MPI_Type_create_f90_complex (       &
     15 , 300 , complextype , error )
```

Searching (1)

You can use global reductions for searching
Bad news: it needs MPI's derived datatypes
Good news: there are some useful built–in ones

We do a reduction with a composite datatype:
(<value> , <index>)

As with sum, we build up from a binary operator
Two built–in operators:
MPI_MINLOC and MPI_MAXLOC
We shall use finding the minimum as an example

Searching (2)

(<value_1>,<index_1>) & (<value_2>,<index_2>)
We shall produce a result (<value_x>,<index_x>)

If <value_1> $\leq$ <value_2> then
    <value_x> $\Leftarrow$ <value_1>
    <index_x> $\Leftarrow$ <index_1>
Else
    <value_x> $\Leftarrow$ <value_2>
    <index_x> $\Leftarrow$ <index_2>

Equality is a bit cleverer – it rarely matters
If it does to you, see the MPI standard

Searching (2)

Operator MPI_MINLOC does precisely that
Operator MPI_MAXLOC searches for the maximum

You create the (<value>,<index>) pairs first
The <index> can be anything – whatever is useful

An <index> should usually be globally unique
I.e. not just the index into a local array
E.g. combine process number and local index

So how do we set up the data?

Fortran Searching (1)

Fortran 77 did not have derived types
The datatypes are arrays of length two

Recommended datatypes are:
MPI_2INTEGER & MPI_2DOUBLE_PRECISION

DOUBLE PRECISION can hold any INTEGER
on any current system, when using MPI

I don't recommend MPI_2REAL, except on Cray

Fortran Searching (2)

```
INTEGER :: sendbuf ( 2 , 100 ) ,        &
       recvbuf ( 2 , 100 ) , myrank , error , i
INTEGER, PARAMETER :: root = 3

DO i = 1 , 100
   sendbuf ( 1 , i ) = <value>
   sendbuf ( 2 , i ) = 1000 * myrank + i
END DO

CALL MPI_Reduce ( sendbuf , recvbuf ,        &
       100, MPI_2INTEGER , MPI_MINLOC ,        &
       root , MPI_COMM_WORLD , error )
```

C Searching

The datatypes are "struct {<value type>; int;}"

- C structure layout is a can of worms

Recommended datatypes are:
MPI_2INT, MPI_LONG_INT, MPI_DOUBLE_INT
For <value type> of int, long and double
That will usually work – not always

Use MPI_LONG_DOUBLE_INT for "long double"

- Don't use MPI_FLOAT_INT or SHORT_INT
  for C reasons you don't want to know!

C Example

```
struct { double value ; int index ; }
       sendbuf [100] , recvbuf [100] ;
int root = 3, myrank , error , i;

for ( i = 1 ; i < 100 ; ++i ) {
    sendbuf [ i ] . value = <value> ;
    sendbuf [ i ] . index = 1000 * myrank + i ;
}

error = MPI_Reduce ( sendbuf , recvbuf ,
       100, MPI_DOUBLE_INT , MPI_MINLOC ,
       root , MPI_COMM_WORLD )
```

Data Distribution (1)

It can be inconvenient to make all counts the same
E.g. with a 100×100 matrix on 16 CPUs

One approach is to pad the short vectors

- That is usually more efficient than it looks

There are also extended MPI collectives for that
Obviously, their interface is more complicated

MPI_Gatherv, MPI_Scatterv,
MPI_Allgatherv, MPI_Alltoallv

- Use whichever approach is easiest for you

Data Distribution (2)

A vector of counts instead of a single count
One count for each process

- Provided only where there are multiple buffers

Receive counts for MPI_Gatherv & MPI_Allgatherv

Send counts for MPI_Scatterv

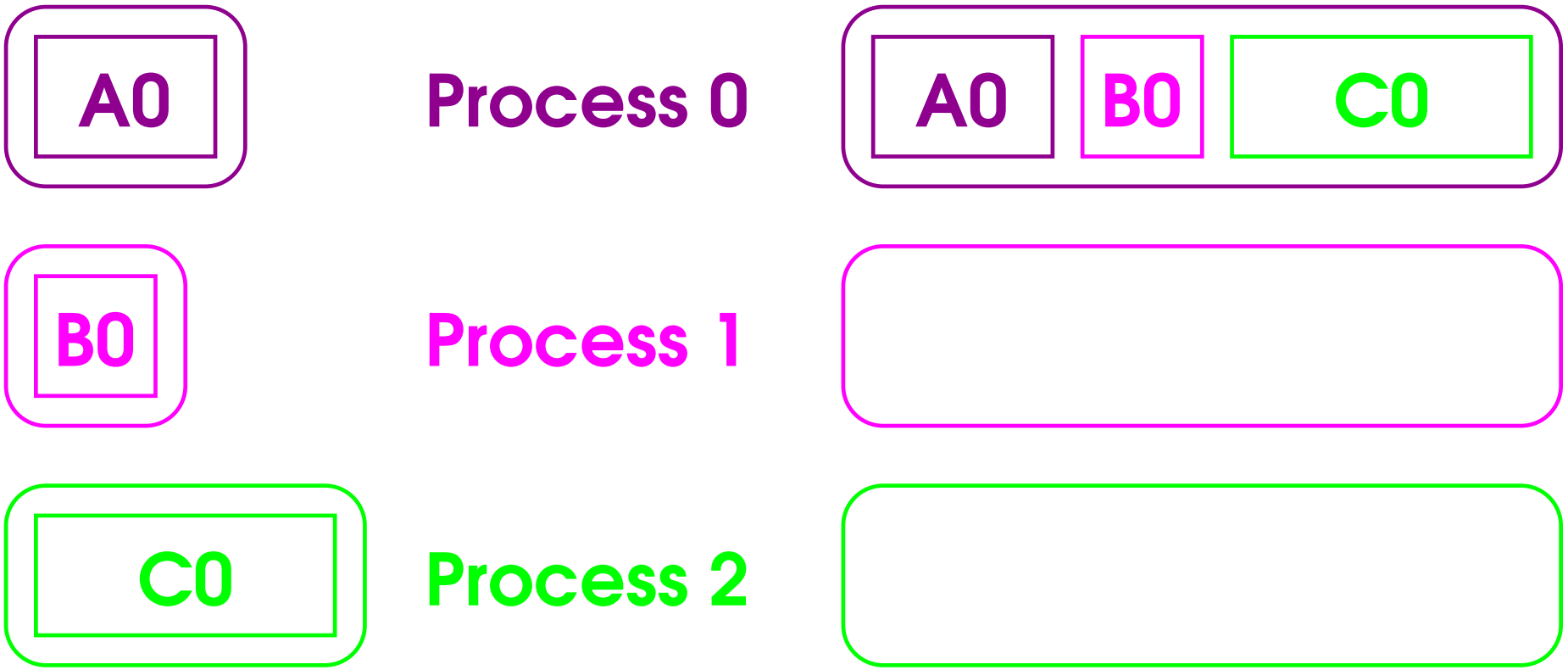
Both counts for MPI_Alltoallv

Used only on root for MPI_Gatherv & MPI_Scatterv

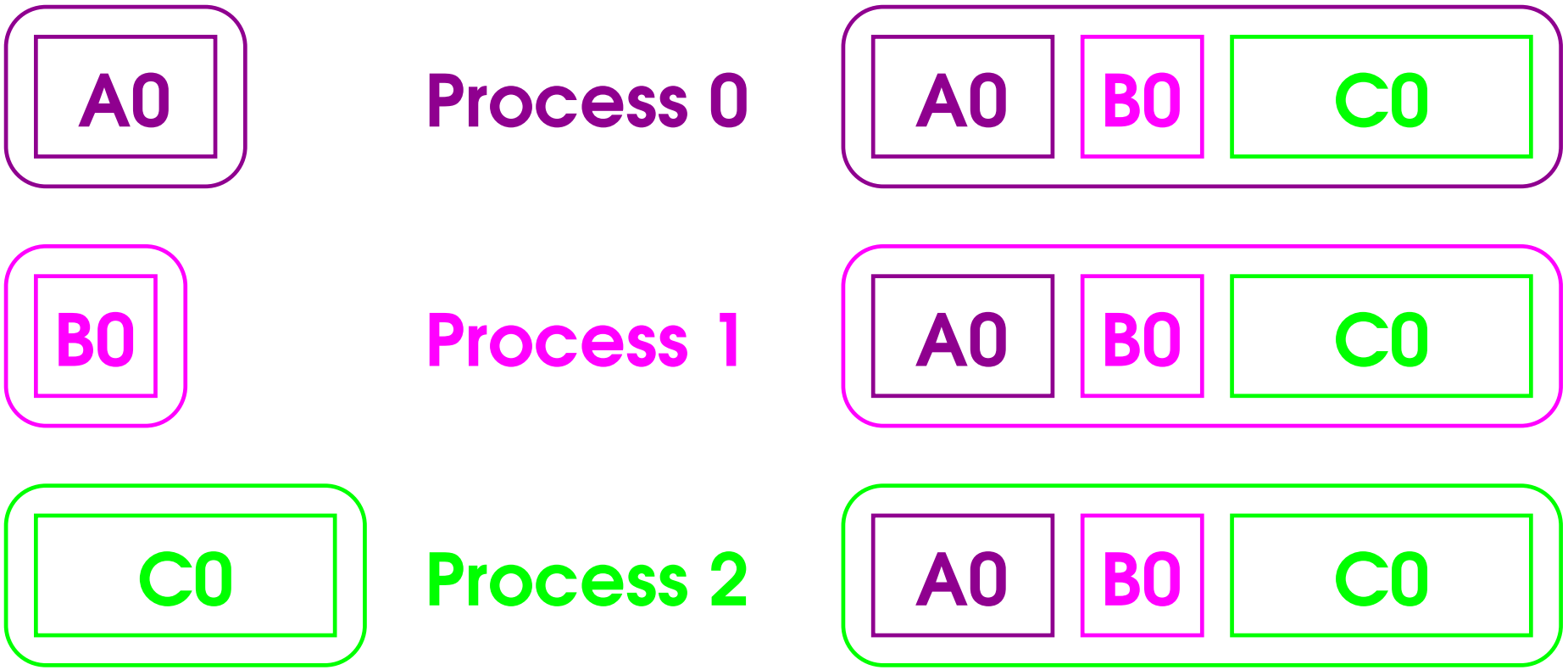
- But, for MPI_Allgatherv and MPI_Alltoallv,
  the count vectors must match on all processes

I recommend always making them match, for sanity

Gatherv



Allgatherv



Data Distribution (3)

The scalar counts may all be different, of course
They must match for each pairwise send and receive

E.g. for MPI_Gatherv:

the send count on process N matches
the Nth receive count element on the root

MPI_Scatterv just the converse of MPI_Gatherv

For MPI_Allgatherv:

the send count on process N matches
Nth receive count element on all processes

Data Distribution (4)

The most complicated one is MPI_Alltoallv

It isn't hard to use, if you keep a clear head

Use a pencil and paper if you get confused

Consider processes M and N

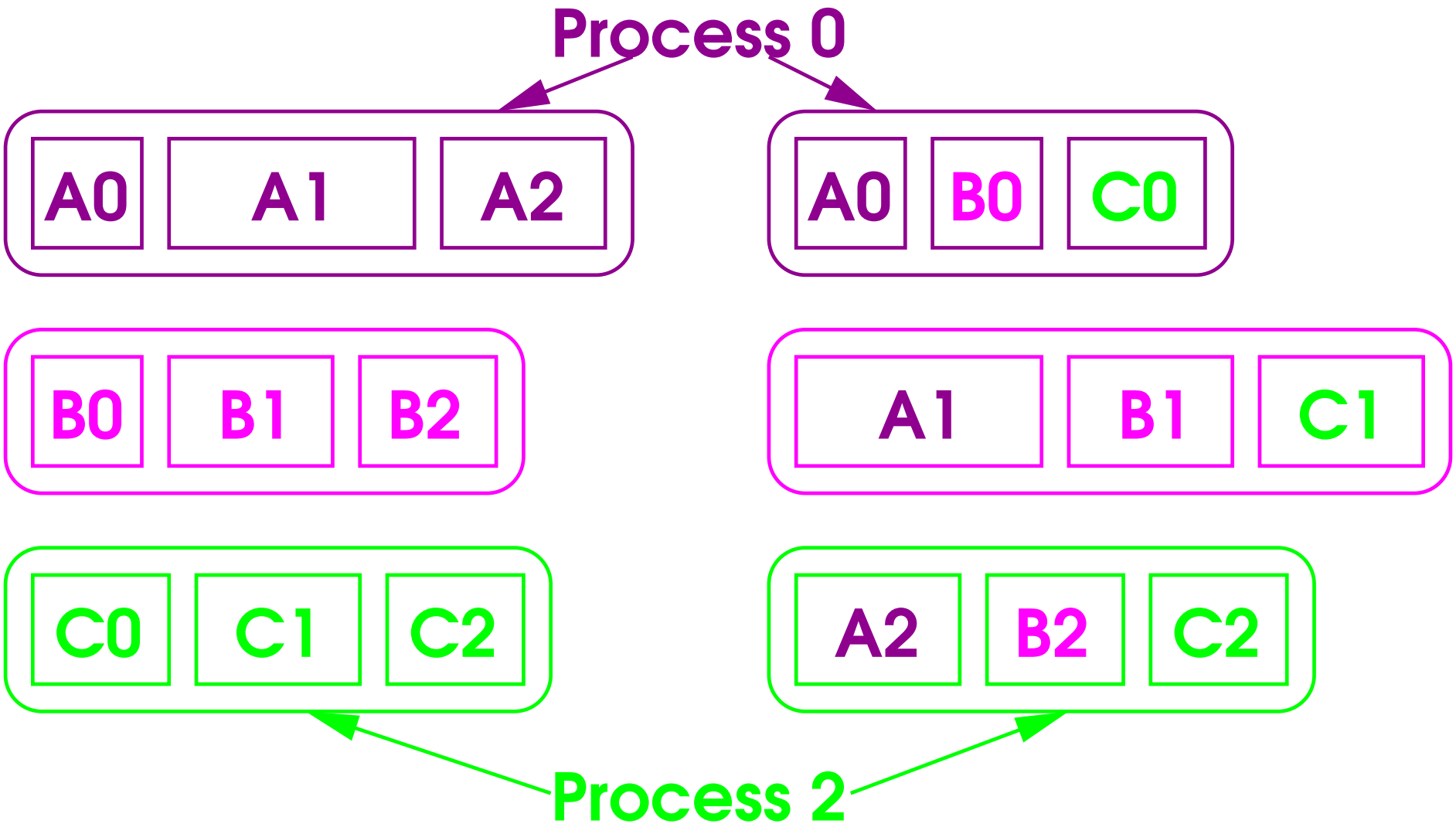
the Nth send count on process M

matches the Mth receive count on process N

As said earlier, think of it as a matrix transpose!

With the data vectors as its elements

Alltoallv


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

Data Distribution (5)

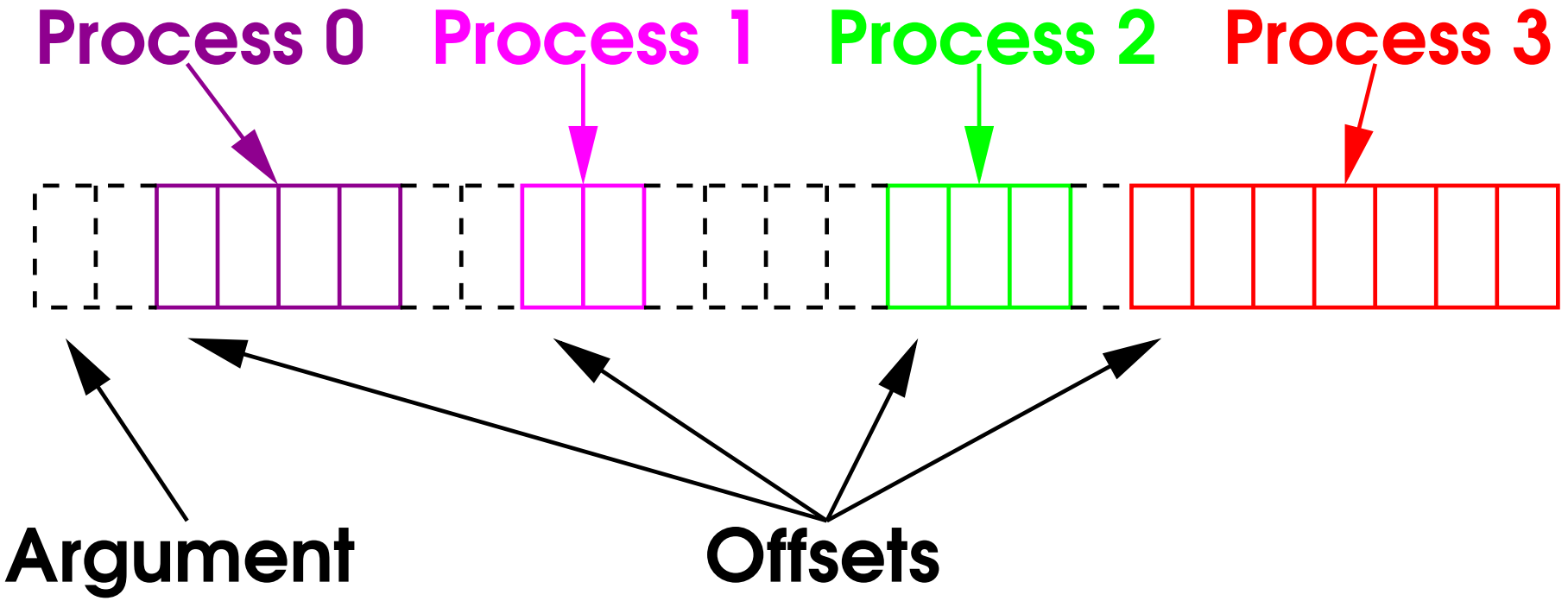
Where they have a vector of counts
they also have a vector of offsets

The offset of the data of each process
Not the offset of the basic elements

This allows for discontiguous buffers
Each pairwise transfer must still be contiguous

Normally, the first offset will be zero
But here is a picture of when it isn't

Multiple Transfer Buffers



Counts = (4 2 3 7)

Offsets = (2 8 14 18)

Data Distribution (6)

Unlike the counts, the offsets are purely local
They need not match on all processes

Even in the case of MPI_Alltoallv
the offset vectors needn't match in any way

Each one is used just as a mapping for
the local layout of its associated buffer

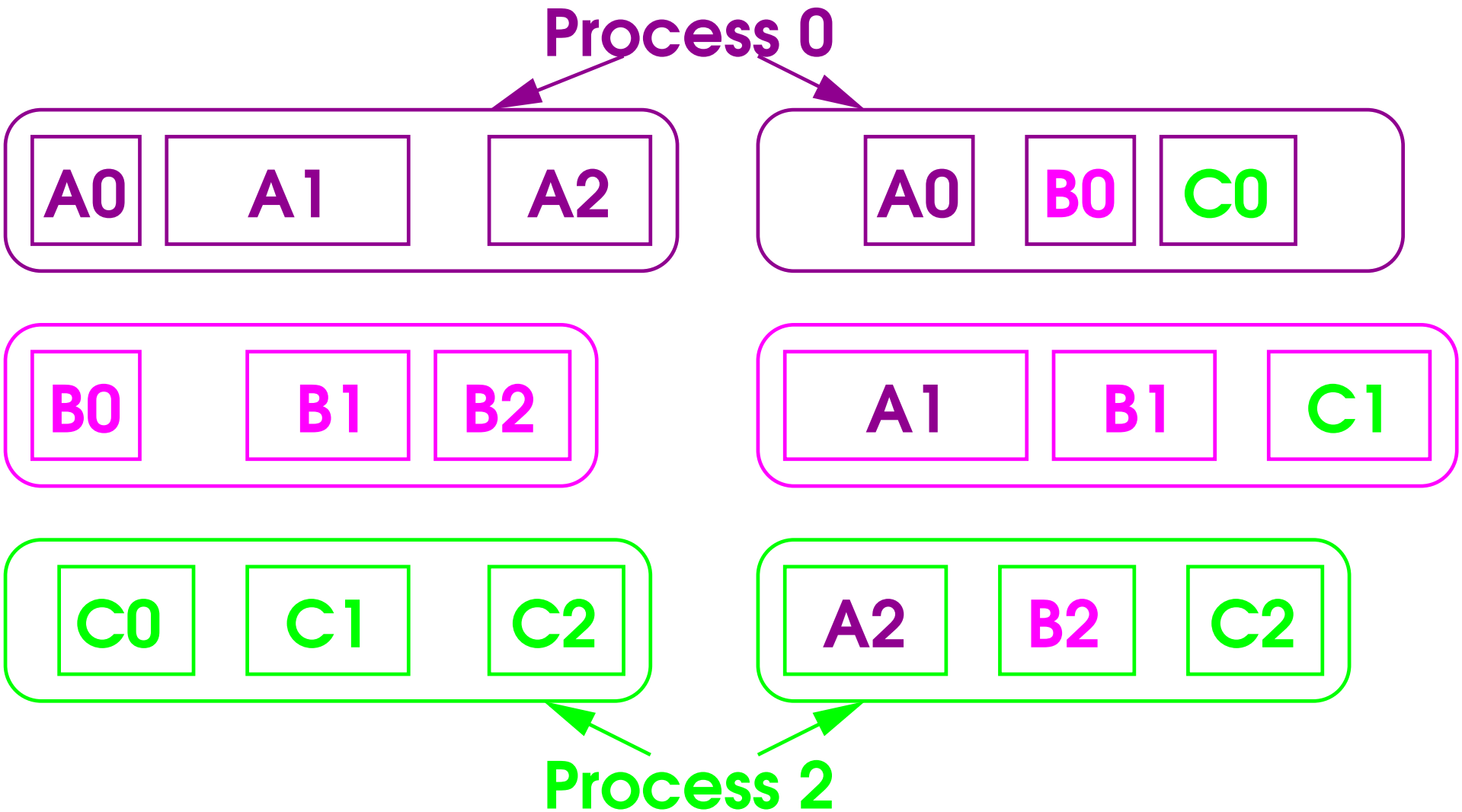
Data Distribution (7)

Keep your use of these collectives simple
MPI won't get confused, but you and I will
And any overlap is undefined behaviour

Will show a picture of a fairly general MPI_Alltoallv
Just to see what can be done, not to recommend it

Then simple examples of using MPI_Gatherv
Start with this (or MPI_Scatterv) when testing

Alltoallv


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

Gatherv (1)

Fortran example:

```
INTEGER , DIMENSION ( 0 : * ) :: counts
REAL(KIND=KIND(0.0D0)) ::        &
    sendbuf ( 100 ) , recvbuf ( 100 , 30 )
INTEGER :: myrank , error, i
INTEGER , PARAMETER :: root = 3 ,        &
    offsets ( * ) = (/ ( 100 * i , i = 0 , 30 - 1 ) /)

CALL MPI_Gatherv ( sendbuf , counts ( myrank ) ,      &
    MPI_DOUBLE_PRECISION ,        &
    recvbuf , counts , offsets ,        &
    MPI_DOUBLE_PRECISION ,        &
    root , MPI_COMM_WORLD , error )
```

Gatherv (2)

C example:

```
int counts [ ] ;
double sendbuf[100] , recvbuf[30][100] ;
int root = 3 , offsets[30] , error, i ;
for ( i = 0 ; i < 30 ; ++ i )
        offsets [ i ] = 100 * i ;

error = MPI_Gatherv (
        sendbuf , counts [ myrank ] , MPI_DOUBLE ,
        recvbuf , counts , offsets ,  MPI_DOUBLE ,
       root , MPI_COMM_WORLD ) ;
```

Practical Point

That's easy when the counts are predictable
But a lot of the time, they won't be

For scatter, calculate the counts on root
Use MPI_Scatter for the count values
Then do the full MPI_Scatterv on the data

Gather calculates a count on each process
Use MPI_Gather for the count values
Then do the full MPI_Gatherv on the data

Allgather and alltoall are similar

Efficiency (1)

- Generally, use collectives whereever possible

Provide most opportunity for implementation tuning

- Use the composite ones where that is useful

  MPI_Allgather, MPI_Allreduce, MPI_Alltoall

- Do as much as possible in one collective

Fewer, larger transfers are always better

Consider packing scalars into arrays for copying
Even converting integers to reals to do so

Efficiency (2)

But what ARE ‘small’ and ‘large’?

- A rule of thumb is about 4 KB per buffer

That is each single, pairwise transfer buffer

But it’s only a rule of thumb and not a hard fact

- Don’t waste time packing data unless you need to

Synchronisation

Collectives are not synchronised at all
except for MPI_Barrier, that is

Up to three successive collectives can overlap
In theory, this allows for improved efficiency

In practice, it makes it hard to measure times

To synchronise, call MPI_Barrier

- The first process leaves only after
the last process enters

Scheduling (1)

Implementations usually tune for gang scheduling
Collectives often run faster when synchronised

- Consider adding a barrier before every collective

It's an absolutely trivial change, after all

Best to run 3–10 times with, and 3–10 without
The answer will either be clear, or it won't matter

If you have major, non–systematic differences
you have a nasty problem, and may need help

Scheduling (2)

You can overlap collectives and point–to–point
MPI requires implementations to make that work

- I strongly advise not doing that

A correct program will definitely not hang or crash
But it may run horribly slowly …

Remember that three collectives can overlap?
Point–to–point can interleave with those

- I recommend alternating the modes of use

It's a lot easier to validate and debug, too!

Scheduling (3)

Start here ...

[Consider calling MPI_Barrier here]

Any number of collective calls

[Consider calling MPI_Barrier here]

Any number of point–to–point calls
Wait for all of those calls to finish

And repeat from the beginning ...

In-Place Collectives (1)

You can usually pass the same array twice
But it's a breach of the Fortran standard
And not clearly permitted in C †

- I recommend avoiding it if at all possible

It will rarely cause trouble – or get diagnosed
But, if it does, the bug will be almost unfindable

I have used systems on which it would fail
The Hitachi SR2201 was one, for example

† Ask me why, offline, if you are masochistic

In-Place Collectives (2)

- MPI 2 defined a MPI_IN_PLACE pseudo–buffer

Specifies the result overwrites the input
I.e. the real buffer is both source and target

Need to read MPI standard for full specification

For allgather[v], alltoall[v] and allreduce[v]:
use it for the send buffer on all processes
Send counts, datatype and offsets ignored

In-Place Collectives (3)

Will give only a C example:

```
        double buffer [ 30 ] [ 100 ] ;
        int error ;
        error = MPI_Alltoall (
                MPI_IN_PLACE , 100 , MPI_DOUBLE ,
/* Or even 'MPI_IN_PLACE , 0 , MPI_INT ,' */
                buffer , 100 , MPI_DOUBLE ,
                MPI_COMM_WORLD )
```

The Fortran is very similar

Epilogue

That is essentially all you need to know!
We have covered everything that seems to be used

MPI collectives look very complicated, but aren't

A few more features, which are rarely used
Mainly introduced by MPI 2 for advanced uses
They are mentioned, briefly, in the extra lectures

Two exercises on searching and scatterv