12 The `parallel sections` Construct

In the following example routines **`XAXIS, YAXIS,`** and **`ZAXIS`** can be executed concurrently. The first **`section`** directive is optional. Note that all **`section`** directives need to appear in the **`parallel sections`** construct.

C/C++

Example 12.1c

```
void XAXIS();
void YAXIS();
void ZAXIS();

void sect_example()
{
  #pragma omp parallel sections
  {
    #pragma omp section
      XAXIS();

    #pragma omp section
      YAXIS();

    #pragma omp section
      ZAXIS();
  }
}
```

C/C++

Fortran

Example 12.1f

```
      SUBROUTINE SECT_EXAMPLE()
!$OMP PARALLEL SECTIONS
!$OMP SECTION
        CALL XAXIS()
!$OMP SECTION
        CALL YAXIS()

!$OMP SECTION
        CALL ZAXIS()

!$OMP END PARALLEL SECTIONS
      END SUBROUTINE SECT_EXAMPLE
```

Fortran

OpenMP - Scheduling(static, dynamic, guided, runtime, auto)

What is Scheduling in OpenMP

Scheduling is a method in OpenMP to distribute iterations to different threads in `for` loop.

The basic form of OpenMP scheduling is

```
#pragma omp parallel for schedule(scheduling-type) for(conditions){
    do something
}
```

Of course you can use `#pragma omp parallel for` directly without scheduling, it is equal to `#pragma omp parallel for schedule(static,1)` [1]

If you run

```
int main()
{
#pragma omp parallel for schedule(static,1)     for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

and

and

```
int main()
{
#pragma omp parallel for        for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

The result stays similar. 20 tasks distributes on 12 threads on my 6-core cpu machine (thread_number = core_number * 2) equally, order to print the result is quite random, but not a big issue(if you run the same code for multiple times, the printed might be different, too)

Result 1:

```
Thread 5 is running number 5
Thread 5 is running number 17
Thread 1 is running number 1
Thread 1 is running number 13
Thread 3 is running number 3
Thread 3 is running number 15
Thread 6 is running number 6
Thread 6 is running number 18
Thread 0 is running number 0
Thread 0 is running number 12
Thread 9 is running number 9
Thread 4 is running number 4
Thread 4 is running number 16
Thread 2 is running number 2
Thread 2 is running number 14
Thread 7 is running number 7
Thread 7 is running number 19
Thread 10 is running number 10
Thread 11 is running number 11
```

```
Thread 8 is running number 8
```

Result 2:

```
Thread 4 is running number 8
Thread 4 is running number 9
Thread 1 is running number 2
Thread 1 is running number 3
Thread 0 is running number 0
Thread 0 is running number 1
Thread 6 is running number 12
Thread 6 is running number 13
Thread 8 is running number 16
Thread 9 is running number 17
Thread 10 is running number 18
Thread 11 is running number 19
Thread 2 is running number 4
Thread 2 is running number 5
Thread 5 is running number 10
Thread 5 is running number 11
Thread 3 is running number 6
Thread 3 is running number 7
Thread 7 is running number 14
Thread 7 is running number 15
```

Static

```
#pragma omp parallel for schedule(static,chunk-size)
```

If you do not specify `chunk-size` variable, OpenMP will divides iterations into chunks that are approximately equal in size and it distributes chunks to threads **in order(Notice that is why static method different from others**.In the `for` loop we discussed before, under 12-thread condition, each thread will treat 1-2 iterations; if you only use 4 threads, each thread will treat 5 iterations.

Result after using `#pragma omp parallel for schedule(static)` (If you do not specify `chunk-size` , the default value is 1)

```
Thread 0 is running number 0
Thread 0 is running number 1
Thread 6 is running number 12
Thread 6 is running number 13
Thread 8 is running number 16
```

```
Thread 3 is running number 6
Thread 3 is running number 7
Thread 2 is running number 4
Thread 2 is running number 5
Thread 9 is running number 17
Thread 10 is running number 18
Thread 11 is running number 19
Thread 5 is running number 10
Thread 5 is running number 11
Thread 1 is running number 2
Thread 1 is running number 3
Thread 4 is running number 8
Thread 4 is running number 9
Thread 7 is running number 14
Thread 7 is running number 15
```

If you specify `chunk-size` variable, the iterations will be divide `into iter_size / chunk_size` chunks.

Notice: iter_size is 20 in this example, because for loop ranges from 0 to 20(not include 20 itself) here

In

```
int main()
{
#pragma omp parallel for schedule(static, 3)    for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

20 iterations will be divided into 7 chunks(6 with 3 iters, 1 with 2 iters), the result is:

```
Thread 5 is running number 15
Thread 5 is running number 16
Thread 5 is running number 17
Thread 2 is running number 6
Thread 2 is running number 7
Thread 2 is running number 8
Thread 6 is running number 18
Thread 6 is running number 19
Thread 1 is running number 3
Thread 1 is running number 4
Thread 1 is running number 5
Thread 3 is running number 9
Thread 3 is running number 10
Thread 3 is running number 11
Thread 4 is running number 12
Thread 4 is running number 13
```

```
Thread 0 is running number 0
Thread 0 is running number 1
Thread 0 is running number 2
Thread 4 is running number 14
```

It is clear that the cpu only uses thread 0, 1, 2, 3, 4, 5, 6 here

But what if `iter_size / chunk_size` is larger than the number of threads in your computer, or number of threads you specified in `omp_set_num_threads(thread_num)` ?

The following example how OpenMP works under this kind of condition.

```
int main()
{
        omp_set_num_threads(4);
#pragma omp parallel for schedule(static, 3)    for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

Result:

```
Thread 1 is running number 3
Thread 1 is running number 4
Thread 1 is running number 5
Thread 1 is running number 15
Thread 1 is running number 16
Thread 1 is running number 17
Thread 3 is running number 9
Thread 3 is running number 10
Thread 3 is running number 11
Thread 0 is running number 0
Thread 0 is running number 1
Thread 0 is running number 2
Thread 0 is running number 12
Thread 0 is running number 13
Thread 0 is running number 14
Thread 2 is running number 6
Thread 2 is running number 7
Thread 2 is running number 8
Thread 2 is running number 18
```

```
Thread 2 is running number 19
```

OpenMP will still split task into 7 chunks, but distributes the chunks to threads **in a circular order**, like the following figure shows

Dynamic

```
#pragma omp parallel for schedule(dynamic,chunk-size)
```

OpenMP will still split task into `iter_size` / `chunk_size` chunks, but distribute trunks to threads dynamically without any specific order.

If you run

```
int main()
{
#pragma omp parallel for schedule(dynamic, 1)   for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

`#pragma omp parallel for schedule(dynamic, 1` is equivalent to `#pragma omp parallel for schedule(dynamic)`

Result:

```
Thread 1 is running number 2
Thread 1 is running number 7
Thread 1 is running number 9
Thread 1 is running number 10
Thread 1 is running number 11
Thread 1 is running number 13
Thread 1 is running number 14
Thread 1 is running number 15
Thread 1 is running number 17
Thread 1 is running number 19
Thread 3 is running number 0
```

```
Thread 0 is running number 4
Thread 8 is running number 12
Thread 4 is running number 3
Thread 6 is running number 6
Thread 9 is running number 16
Thread 5 is running number 1
Thread 7 is running number 8
Thread 10 is running number 18
Thread 2 is running number 5
```

You can see that thread 1 took on 10 iters while others took only 0-1.

Comparing with `static` Method:

Pros: The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are not as balance as static method between each other.

Cons: The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.[1]

Guided

```
#pragma omp parallel for schedule(guided,chunk-size)
```

Chunk size is dynamic while using guided method, the size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads, and the size will be decreased to `chunk-size` (but the last chunk could be smaller than `chunk-size`)

Use a 4-thread structure to see what will happen in a 20-iter `for` loop after applying guided method:

```
int main()
{
        omp_set_num_threads(4);
#pragma omp parallel for schedule(guided, 3)    for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
```

OpenMP Reductions

```
double avg = 0.0;
double A[SIZE];
#pragma  omp  parallel  for
for (int i = 0; i < SIZE; i++;)
{
  avg += A[i];
 }
avg = avg / SIZE;
```

- Reductions commonly occur in codes (as in pi example)
- OpenMP provides special support via “reduction” clause
  - OpenMP compiler automatically creates local variables for each thread, and divides work to form partial reductions, and code to combine the partial reductions
  - Predefined set of associative operators can be used with reduction clause, e.g., +, *, -, min, max

OpenMP Reductions

```
double avg = 0.0;
double A[SIZE];
#pragma  omp  parallel  for reduction(+ : avg)

for (int i = 0; i < SIZE; i++;)
{
  avg += A[i];
 }
avg = avg / SIZE;
```

- Reductions clause specifies an operator and a list of reduction variables (must be shared variables)
  - OpenMP compiler creates a local copy for each reduction variable, initialized to operator's identity (e.g., 0 for +; 1 for *)
  - After work-shared loop completes, contents of local variables are combined with the "entry" value of the shared variable
  - Final result is placed in shared variable

Installing MPI in Linux

This document describes the steps used to install MPICH2, the MPI-2 implementation from Argonne National Laboratory in UNIX (Fedora Core 4) based system. Most of the steps followed here, have been explained in MPICH2 Installer's Guide which is the source of this document.

We need the following prerequisites

1) The tar file mpich2-1.0.5p3.tar.gz (which can be obtained from http://www-unix.mcs.anl.gov/mpi/mpich2/)
2) A C compiler (gcc is sufficient)
3) A Fortran compiler if Fortran applications are to be used (g77 is sufficient)

Both the C and Fortran compiler are present in Fedora Core 4 by default.

Step 1. Create a directory MPI(we can use any name) in the home directory.

```
$ cd $HOME
$ mkdir MPI
```

Step 2. Unpack the tar file.

```
$ tar xfz  mpich2-1.0.5p3.tar.gz
```

The directory MPI will now contain a sub-directory mpich2-1.0.5p3.

Step 3. Choose an installation directory (the default is /usr/local/bin)

```
$ mkdir mpich2-install
```

Step 4. Choose a build directory

```
$ mkdir mpich2-1.0.5
```

Now the MPI directory will contain three sub-directories namely mpich2-1.0.5p3, mpich2-1.0.5 and mpich2-install.

Step 5. Configure MPICH2, specifying the installation directory and running the configure script in the source directory.

```
$ cd $HOME
$ cd MPI/mpich2-1.0.5
$/home/you/MPI/mpich2-1.0.5p3/configure -prefix=/home/you/MPI/mpich2-install
```

For other configure options please refer the MPICH2 Installer's Guide

Step 6. Build MPICH2

```
$ make
```

Step 7. Install the MPICH2 commands.

$ make install

Step 8. Add the bin directory to your path.

$ export PATH=/home/you/MPI/mpich2-install/bin:$PATH

(It is better to add this line in .bash_profile file present in the home directory so that this path gets permanently added once we reboot the system.

$ cd $HOME
$ vi **.bash_profile**
Then append the above command of step 8.)

We can check that everything is in order at this point by doing

$ which mpd
$ which mpicc
$ which mpiexec
$ which mpirun

All should refer to the commands in the bin subdirectory of our install directory.

The MPI has been successfully installed now. We can follow the same steps to install MPI in other machines.

We will have to follow these steps to form a cluster using MPI.

Step 1. We must have a valid host name for each system. The host name localhost.localdomain is not accepted by MPI. We can change the host name by

$ vi **/etc/sysconfig/network**

Step 2. Now we must add the IP address of home machine and other machines in /etc/hosts

$vi **/etc/hosts**

We will get a file like this

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1       localhost.localdomain   localhost
```

Here we can add the IP address and hostname's of other machines. Remember to put the home machine at the top. The file /etc/hosts after appending looks like this.

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1       localhost.localdomain   localhost
172.16.8.75     system1
172.16.8.76     system2
172.16.5.46     system3
172.16.5.43     system4
```

After these two steps please type the following command else the system will take a lot of time to boot.

$ chkconfig sendmail off

Step 3. Create a file consisting of a list of machine names, one per line. Name this fille **mpd.hosts**.

```
$ cd $HOME
$ vi mpd.hosts
```

These three steps should be followed in all the machines.

The users before starting MPI programs must follow the steps given below.

Step 1. On each system, only once we have to execute these commands.

```
$ touch .mpd.conf
$ chmod 600 .mpd.conf
$ vi .mpd.conf
```

and add

MPD_SECRETWORD=<_____> any word without spaces, which should be same for each user in all the systems.

We should do this so that one user daemon should not communicate with daemons of other users. This mechanism is a must, without which your MPI interface would not start.

Step 2. To coordinate the activities between different systems for each user, each user will be running a daemon named 'mpd' on each system. Among them one daemon will be awarded as Master Daemon, which to be initiated at first. Then other mpd daemons of other nodes are started later as will be shown in step 3.

```
$ mpd &
$ mpdtrace -l
```

It gives hostname_portno (IP address) as the output. Note down the portno, and have to provide to other mpd daemons, which to be run on other systems.

Step 3. Login into other systems, and start mpd daemons as such

```
$ mpd -h <hostname of master daemon> -p <portno> &
```

Step 4. Run

```
mpdtrace -l
```

to check how many systems are included in our loop. It differs for every user, because every user has to launch his/her respective daemons on their chosen nodes.

Step 5. Check your personal multi-processor environment by

```
mpiexec -n 4 /bin/hostname
```

We can now write our MPI programs and Compile any program using MPI constructs by

```
mpicc -o <object file> <source code>.c
```

Execute them on our formulated MPI environment by

```
mpiexec/mpirun -np <i> <object file>
```

Please also refer to the documents MPICH2 Installer's Guide and User's Guide available at http://www-unix.mcs.anl.gov/mpi/mpich/

Programming with MPI

Datatypes and Collectives

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

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/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/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/C++: `int buffer [ 100 ] ;`

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/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 few 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

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

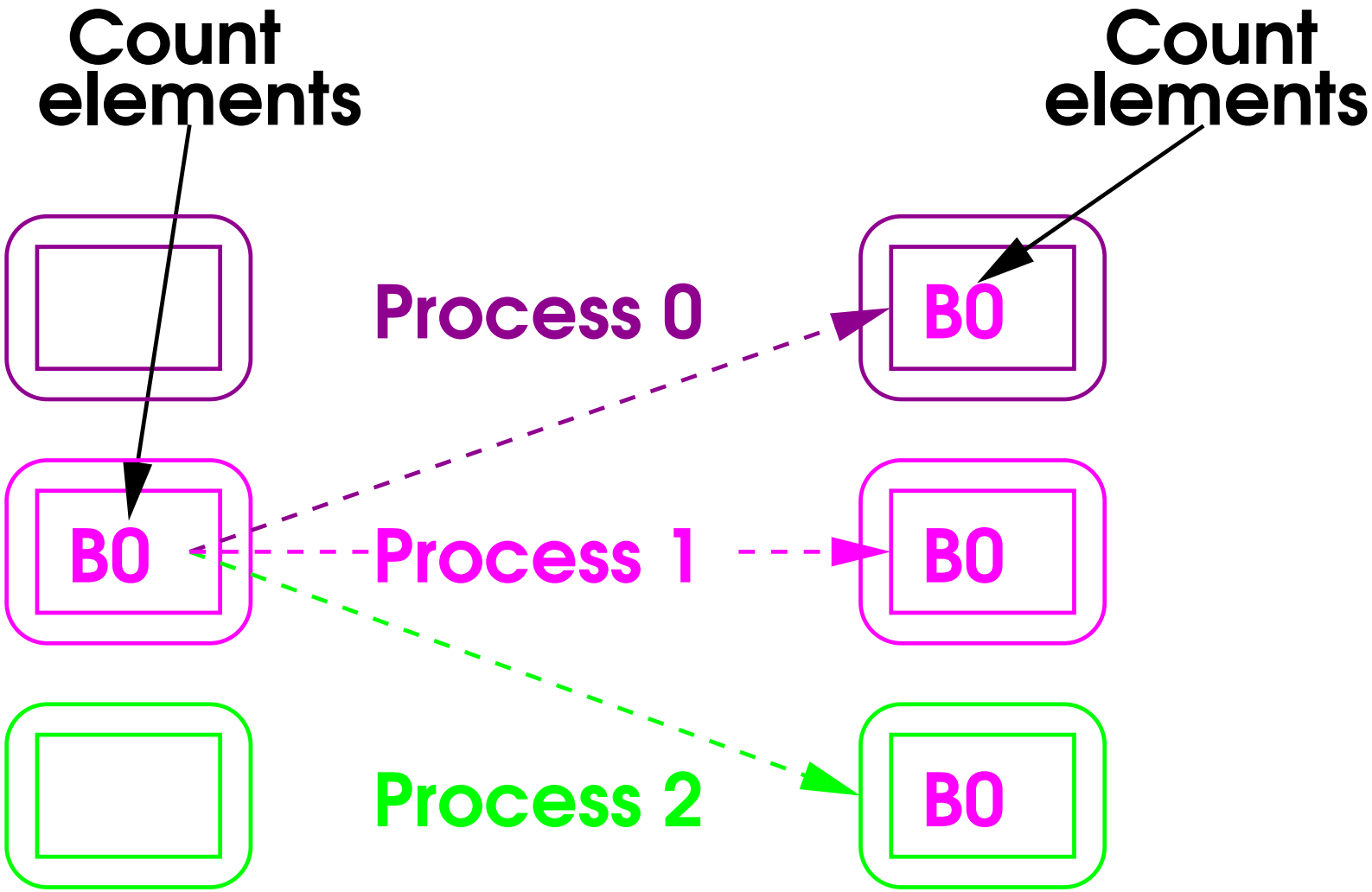
Needed for advanced features not covered here

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 processors 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++ example:

```
double buffer [ 100 ];
int root = 3 ;
MPI::COMM_WORLD . Bcast ( buffer , 100 ,
      MPI::DOUBLE , root ) ;
```

Multiple Transfer Buffers

Many collectives need one buffer per process

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

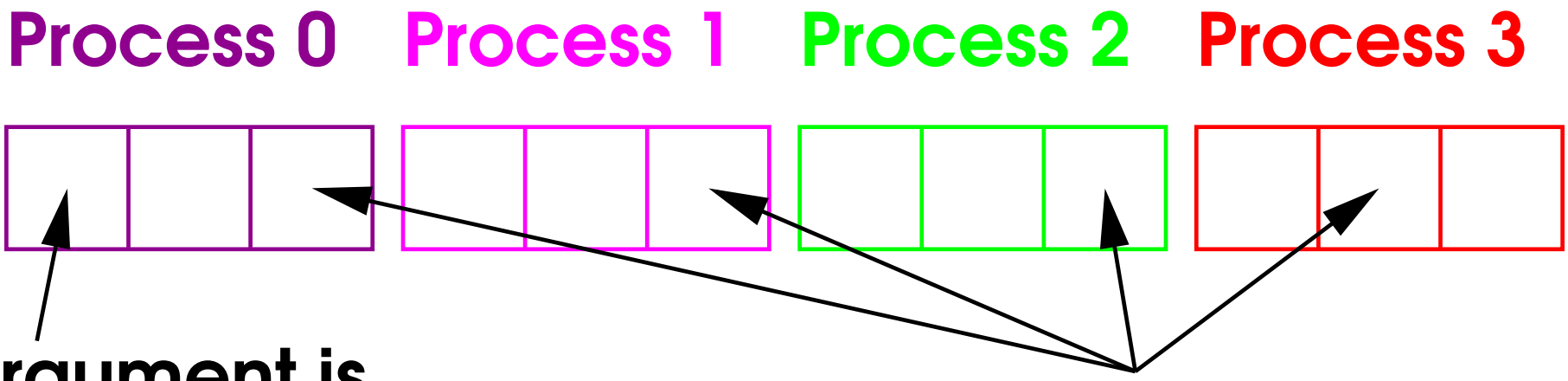
Each pairwise transfer buffer is concatenated
in the order of process numbers (i.e. $0 \ldots N-1$)

Size of source = $N *$ size of each result

Multiple Transfer Buffers

This is for 4 processes

A count (vector length) of 3



Argument is address of first element (as usual)

Elements (i.e. one unit of the datatype)

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/C++

Nor does ‘‘<type> :: buffer(*)’’ in Fortran

- It is up to you to get it right

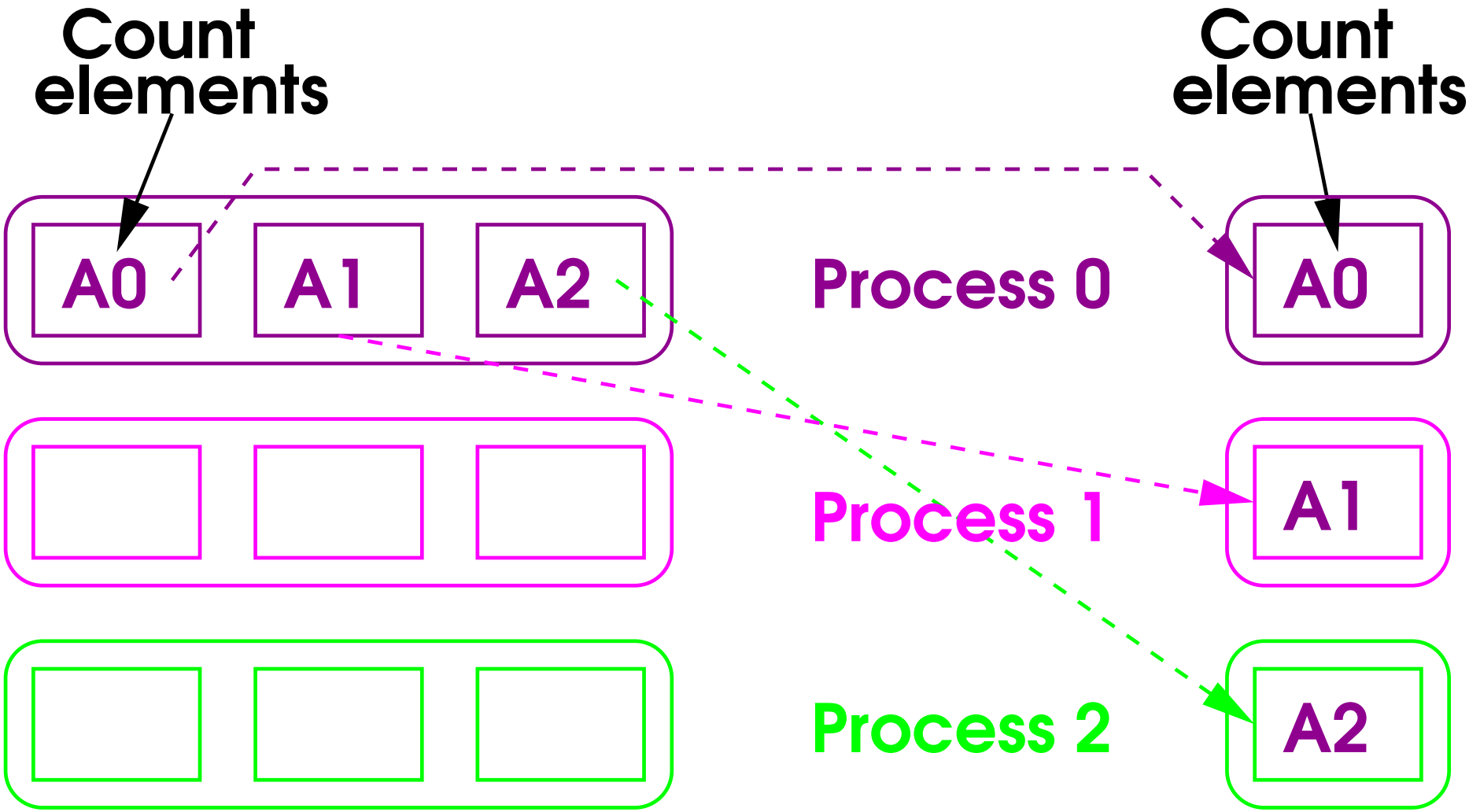
No compiler can trap an error with that

We shall use scatter as our first example

This is one process sending different data

to every process in the communicator

Scatter


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

Scatter (1)

Scatter copies different data from the root
to all processors 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:

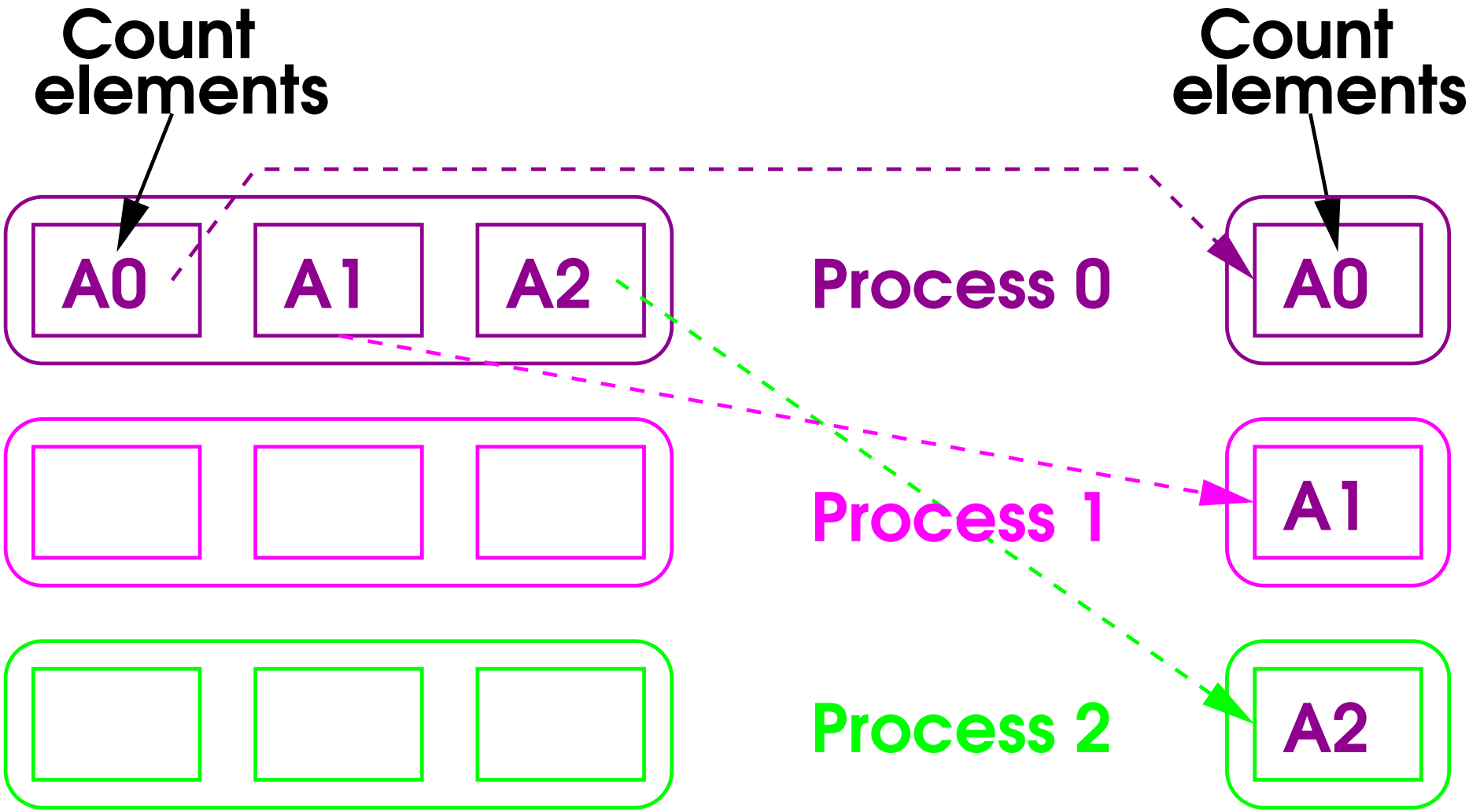
```
double sendbuf [ 30 ] [ 100 ] , recvbuf [ 100 ] ;
int root = 3 , error ;
error = MPI_Scatter (
        sendbuf , 100 , MPI_DOUBLE ,
        recvbuf , 100 , MPI_DOUBLE ,
        root , MPI_COMM_WORLD )
```

Scatter (4)

C++ example:

```
double sendbuf [ 30 ] [ 100 ] , recvbuf [ 100 ] ;
int root = 3 ;
MPI::COMM_WORLD . Scatter (
        sendbuf , 100 , MPI::DOUBLE ,
        recvbuf , 100 , MPI::DOUBLE ,
        root )
```

Scatter


Count elements
Count elements
A0
A1
A2
Process 0
A0
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 (≡ 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)

MPI–2 supports Fortran 90 parameterized types
REAL(KIND=SELECTED_REAL_KIND(15,300))
There is more on those in the extra lectures

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/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–2 only)
- MPI_SHORT
- MPI_UNSIGNED_SHORT
- MPI_INT
- MPI_UNSIGNED (not MPI_UNSIGNED_INT)
- MPI_LONG
- MPI_UNSIGNED_LONG

C/C++ Datatypes (2)

Recommended floating–point datatypes:

MPI_FLOAT

MPI_DOUBLE

MPI_LONG_DOUBLE

For use from C/C++, I recommend one more

MPI_BYTE (uninterpreted 8–bit bytes)

What you can do with these is a bit restricted

- Remember MPI_ in C is MPI:: in C++

Though the C names may well be accepted in both

C++ Datatypes

Recommended datatypes (in C++ but not C) :

MPI::BOOL
MPI::COMPLEX
MPI::DOUBLE_COMPLEX
MPI::LONG_DOUBLE_COMPLEX

They all correspond to the obvious C++ type

Other C/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

Both C99 and MPI–2 and optional, anyway

MPI_WCHAR (whatever C/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 types

- Ask me offline why that is a Good Thing

Gather

Gather is precisely the converse of scatter

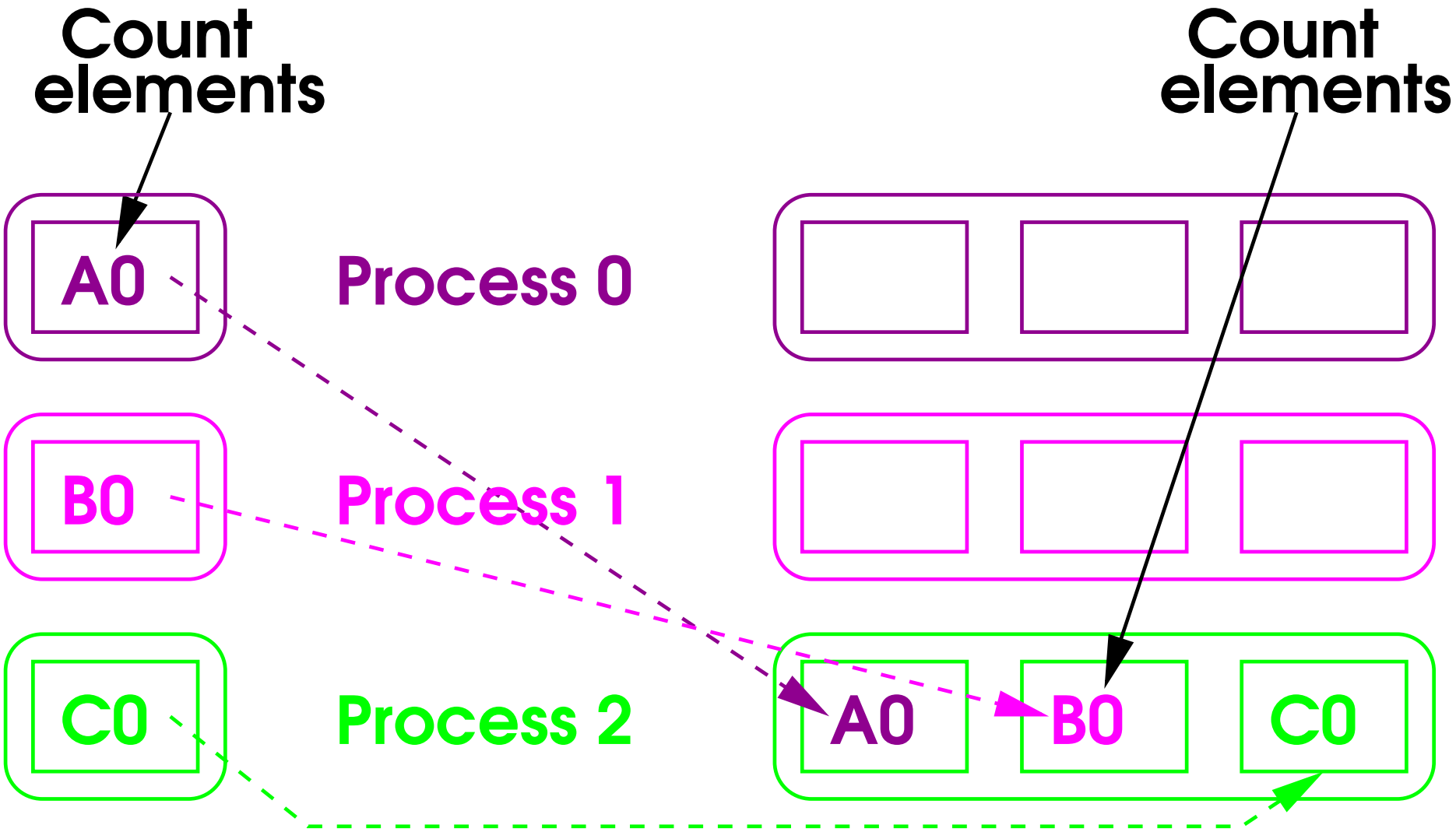
- Just change MPI_Scatter to MPI_Gather
  And Scatter to Gather for C++, of course

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

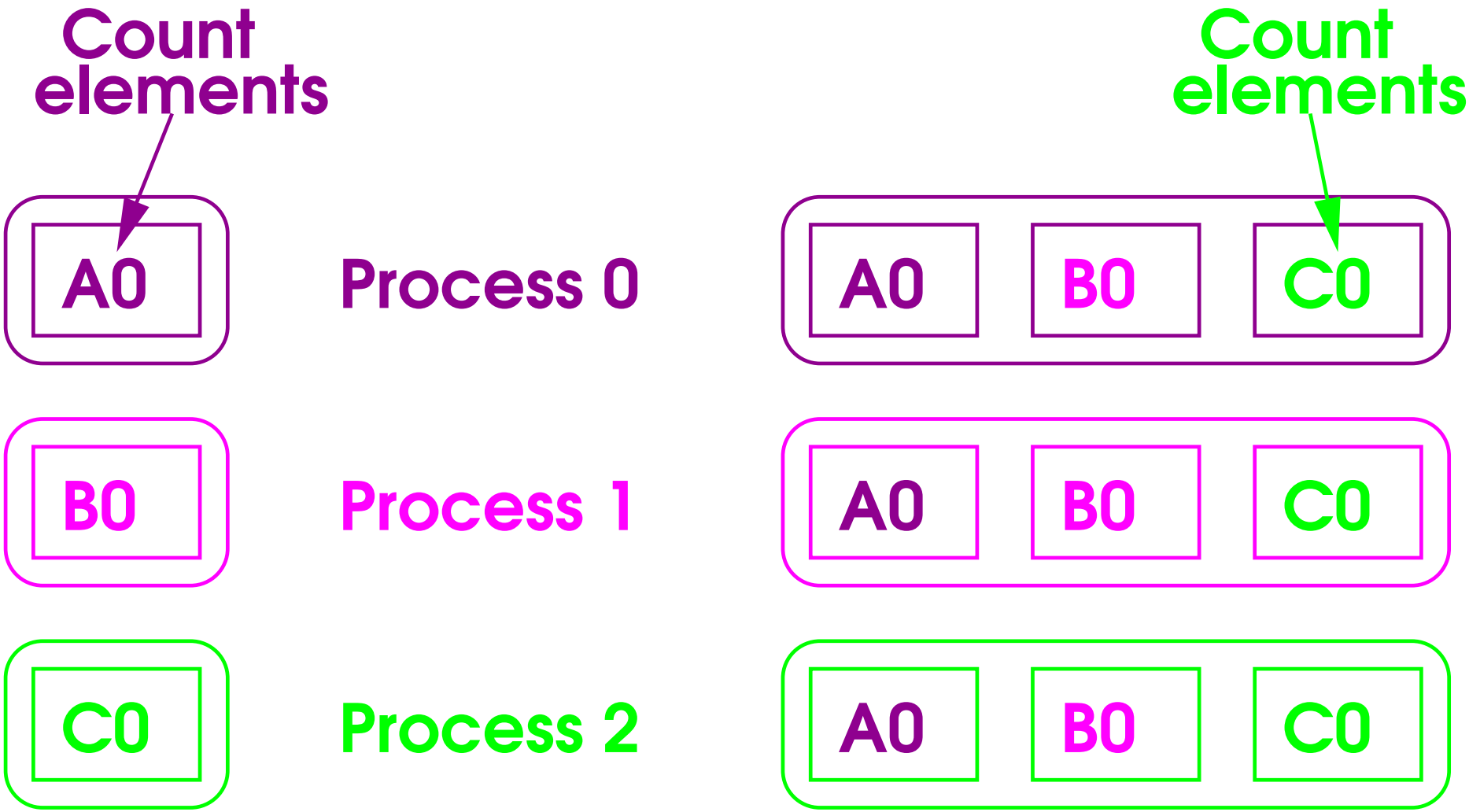
- Change MPI_Gather to MPI_Allgather

  And Gather to Allgather for C++

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
B0
Process 1
C0
Process 2
A0
B0
C0
A0
B0
C0
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 )
```

C++ example:

```
double sendbuf [ 100 ] , recvbuf [ 30 ] [ 100 ] ;
MPI::COMM_WORLD . Allgather (
        sendbuf , 100 , MPI::DOUBLE ,
        recvbuf , 100 , MPI::DOUBLE )
```

Alltoall

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

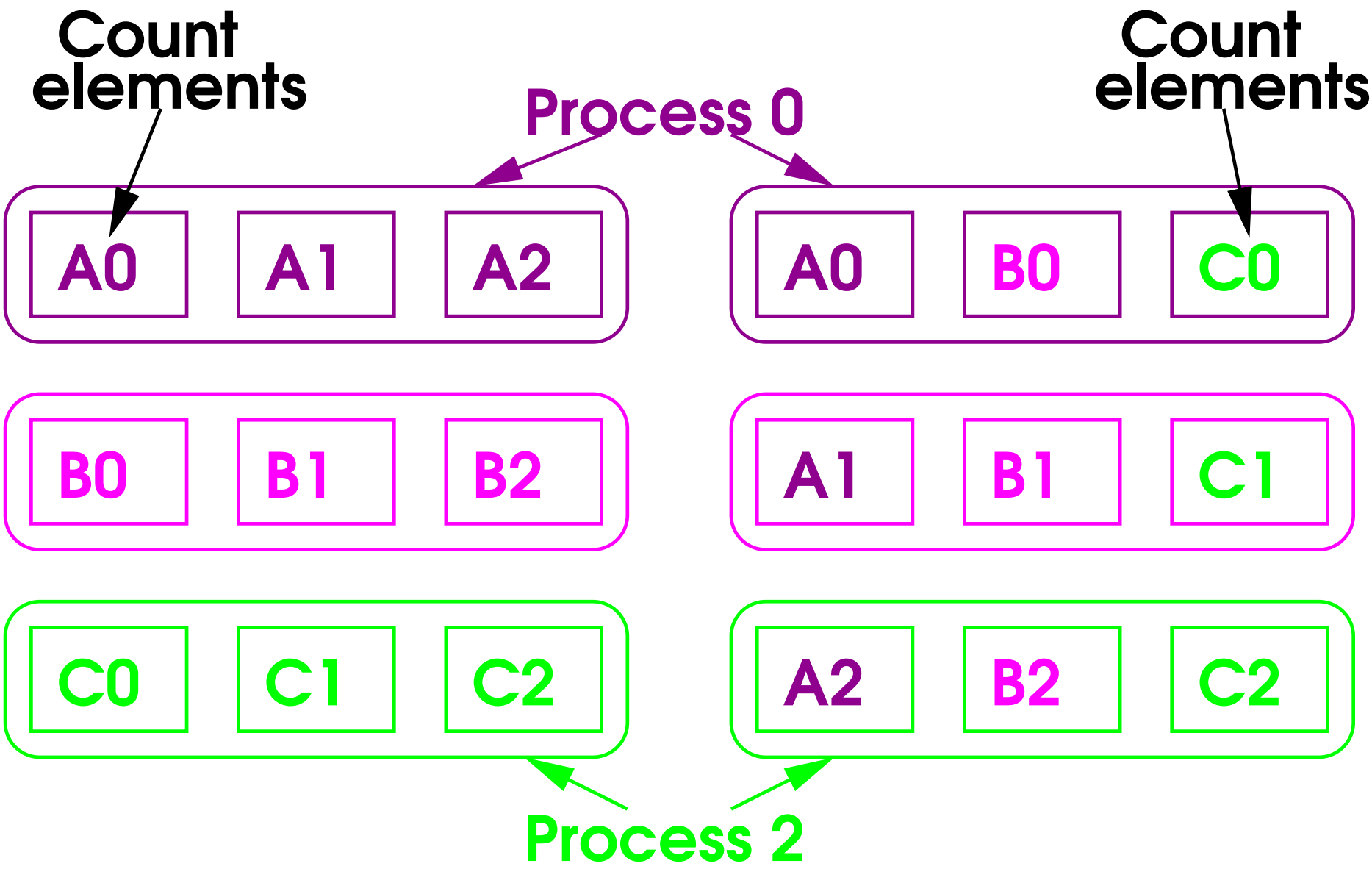
- Just change MPI_Allgather to MPI_Alltoall
  And Allgather to Alltoall for C++

- 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)

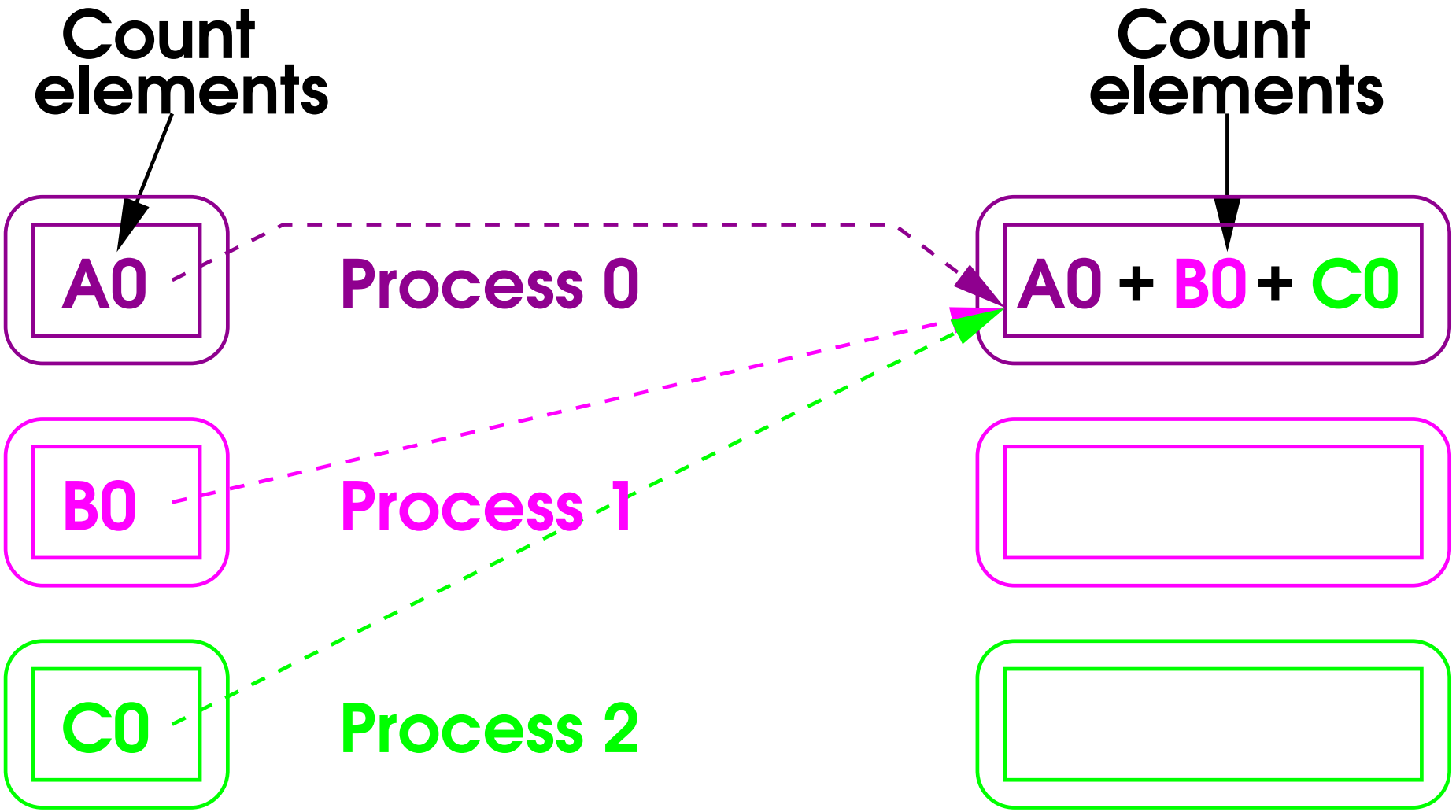
One of the basic parallelisation primitives

Start with a normal gather operation
Then sum the values over all processors
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

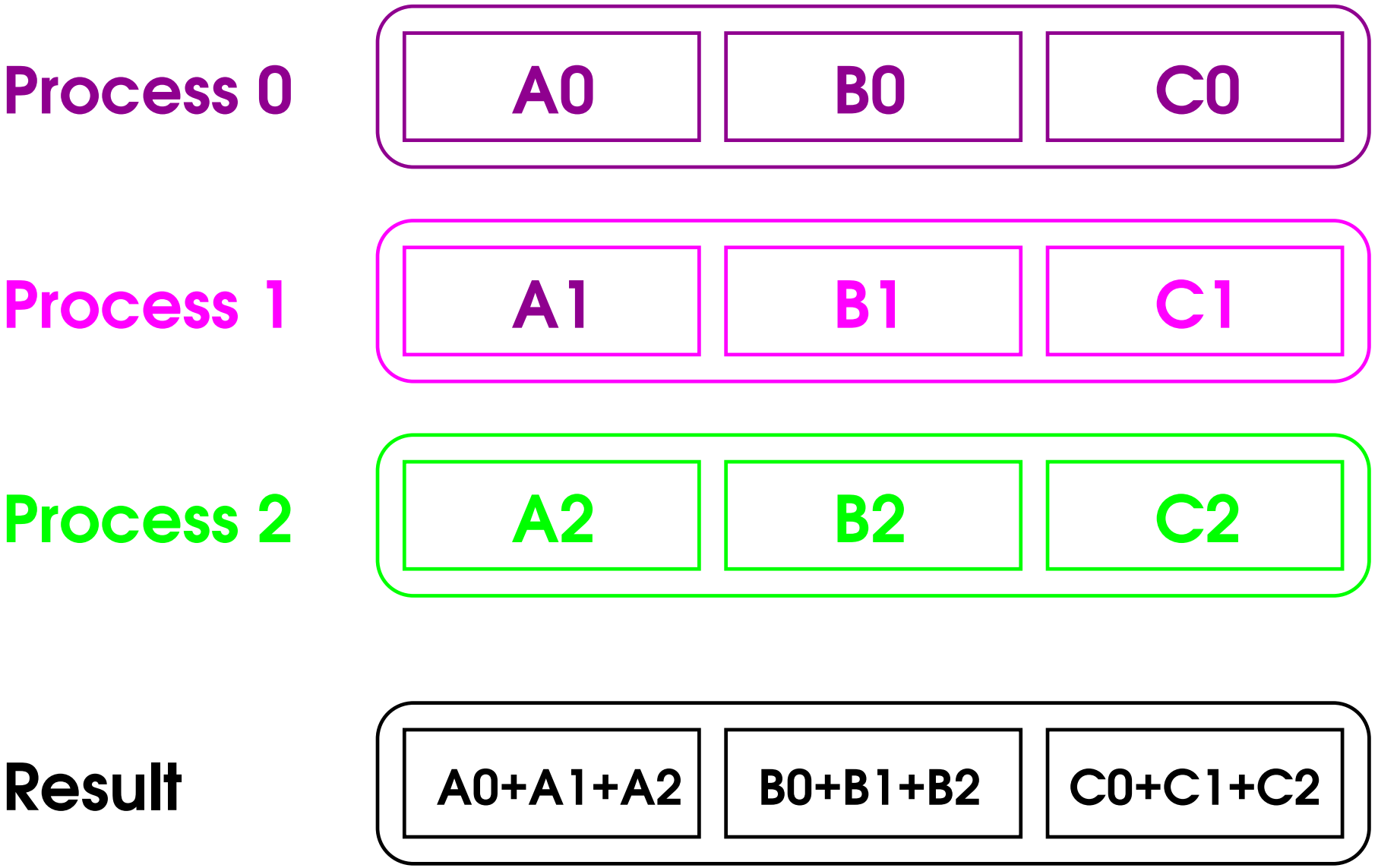
- Does not reduce over the vector

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 )
```

C++ example:

```
double sendbuf [ 100 ] , recvbuf [ 100 ] ;
int root = 3 ;
MPI::COMM_WORLD . Reduce (
      sendbuf , recvbuf , 100 , MPI::DOUBLE ,
      MPI::SUM , root )
```

Allreduce

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

- This is now a symmetric operation

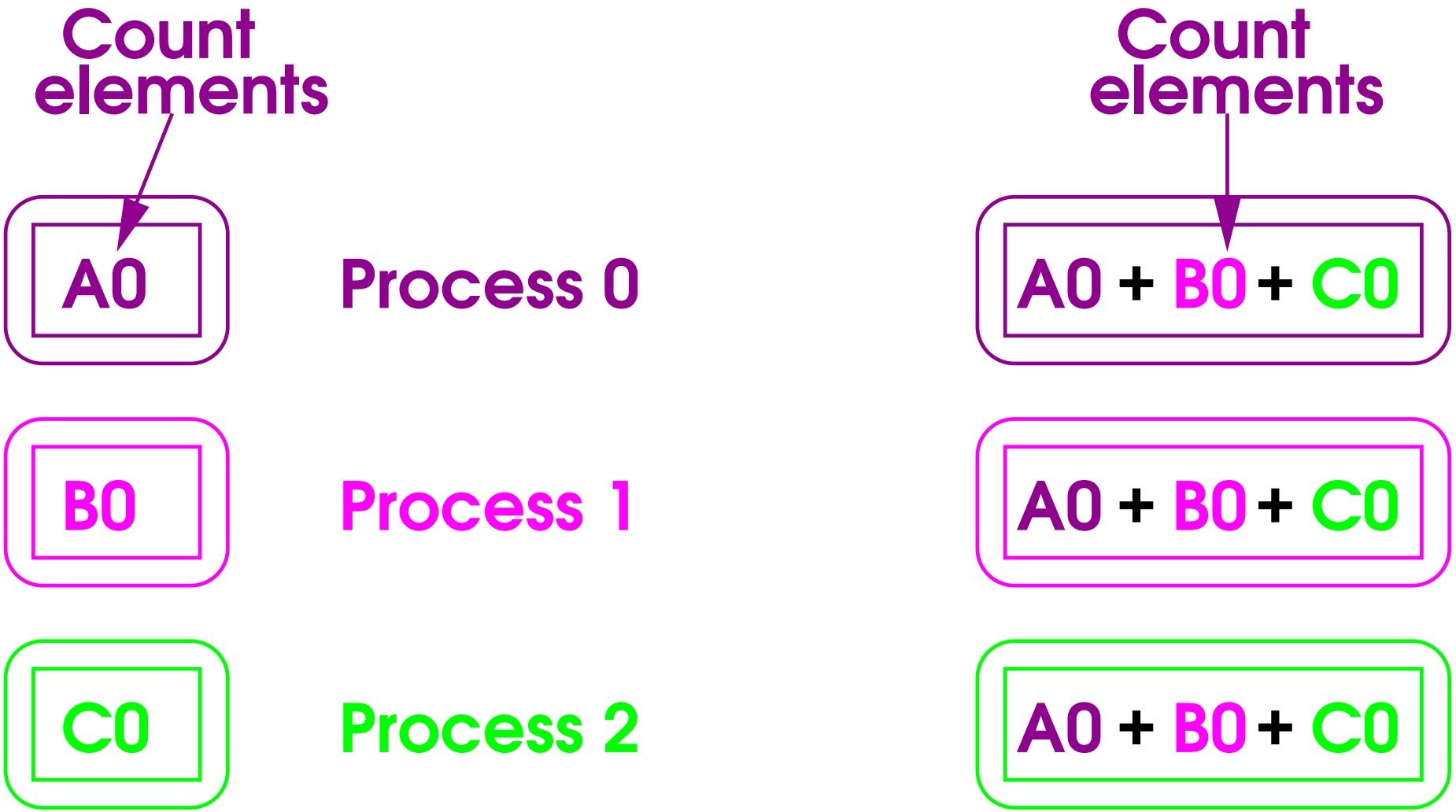
So has no argument specifying the root process

- Just change MPI_Reduce to MPI_Allreduce
  - And Reduce to Allreduce for C++

And remove the root process argument, of course

- The receive buffer is now used on all processes

Allreduce


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

Reduction Operations (1)

Remember the C++ name changes
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/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:

- A few more advanced collectives
  Searching as a reduction
  More flexible buffer layout

- 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

Dated :
Assessment No. : 7

Basic MPI Communication Routines

AIM

Consider the following program, called mpi_sample1.c. This program is written in C with MPI commands included.

The new MPI calls are to MPI_Send and MPI_Recv and to MPI_Get_processor_name. The latter is a convenient way to get the name of the processor on which a process is running. MPI_Send and MPI_Recv can be understood by stepping back and considering the two requirements that must be satisfied to communicate data between two processes:

1. Describe the data to be sent or the location in which to receive the data
2. Describe the destination (for a send) or the source (for a receive) of the data.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]){
        int  my_rank; /* rank of process */
        int  p;       /* number of processes */
        int source;   /* rank of sender */
        int dest;     /* rank of receiver */
        int tag=0;    /* tag for messages */
        char message[100];        /* storage for message */
        MPI_Status status ;   /* return status for receive */
        /* start up MPI */
        [redacted]
        /* find out process rank */
        [redacted]
        /* find out number of processes */
        [redacted]
[redacted]
                /* create message */
                [redacted]
[redacted]
                /* use strlen+1 so that '\0' get transmitted */
                [redacted]
[redacted]
        }
        else{
                [redacted]
[redacted]
                }
        }
        /* shut down MPI */
        [redacted]
        return 0;
}
```

1. Implement the above code
2. Build and Execute the logical scenario with few test cases
3. Depict the screenshots along with proper justification

Collective communications

MPI Broadcast

The same data is sent from the root to all processes in the communicator

In C:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype data_type, int root, MPI_Comm comm);
```

In Fortran:

```
call MPI_Bcast(buffer, count, data_type, root, comm, error)
```

MPI Scatter

Different data is sent to each process in the communicator

In C:

```
MPI_Scatter(void *sendbuffer, int sendcount, MPI_Datatype senddata_type,
            void *recvbuffer, int recvcount, MPI_Datatype recvdata_type,
            int root, MPI_Comm comm);
```

In Fortran:

```
call MPI_SCATTER(sendbuffer, sendcount, senddata_type,
                 recvbuffer, recvcount, recvdata_type,
                 root, comm, error);
```

Collective communications

Example: code demonstrating Broadcast subroutine

Look for "**broadcast.c**"

C example:

```
# include <mpi.h>
Int main (int argc, char *argv[])
{
        int rank;
        double param;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if(rank==2) param=23.0;
        MPI_Bcast(&param,1,MPI_DOUBLE,2,MPI_COMM_WORLD);
        printf("P:%d after broadcast parameter is %f\n",rank,param);
        MPI_Finalize();
}
```

Collective communications

Example: code demonstrating Broadcast subroutine

Fortran example:

Look for "**broadcast.f**"

```
program BROADCAST
include 'mpif.h'
integer error, rank, size
real param
call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,error)
print*,"P:", rank, "after broadcast param is ", param
call MPI_FINALIZE(error)
end
```

Collective communications

Example: code demonstrating Broadcast subroutine

Running “broadcast.c” :

mpirun -np 4 ./broadcast.exe

Output:

P:0 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000

Gather

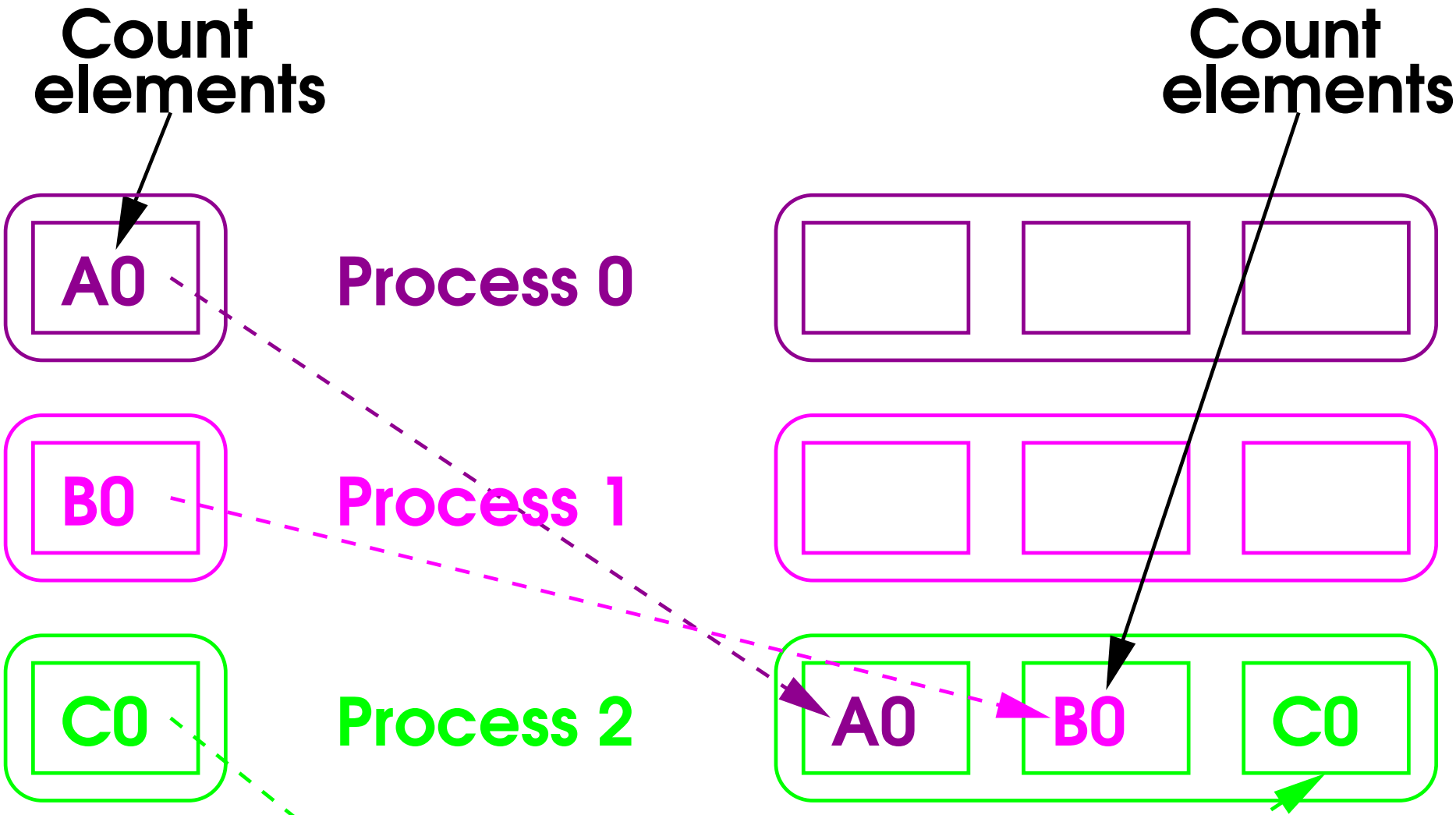
Gather is precisely the converse of scatter

- Just change MPI_Scatter to MPI_Gather
  And Scatter to Gather for C++, of course

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

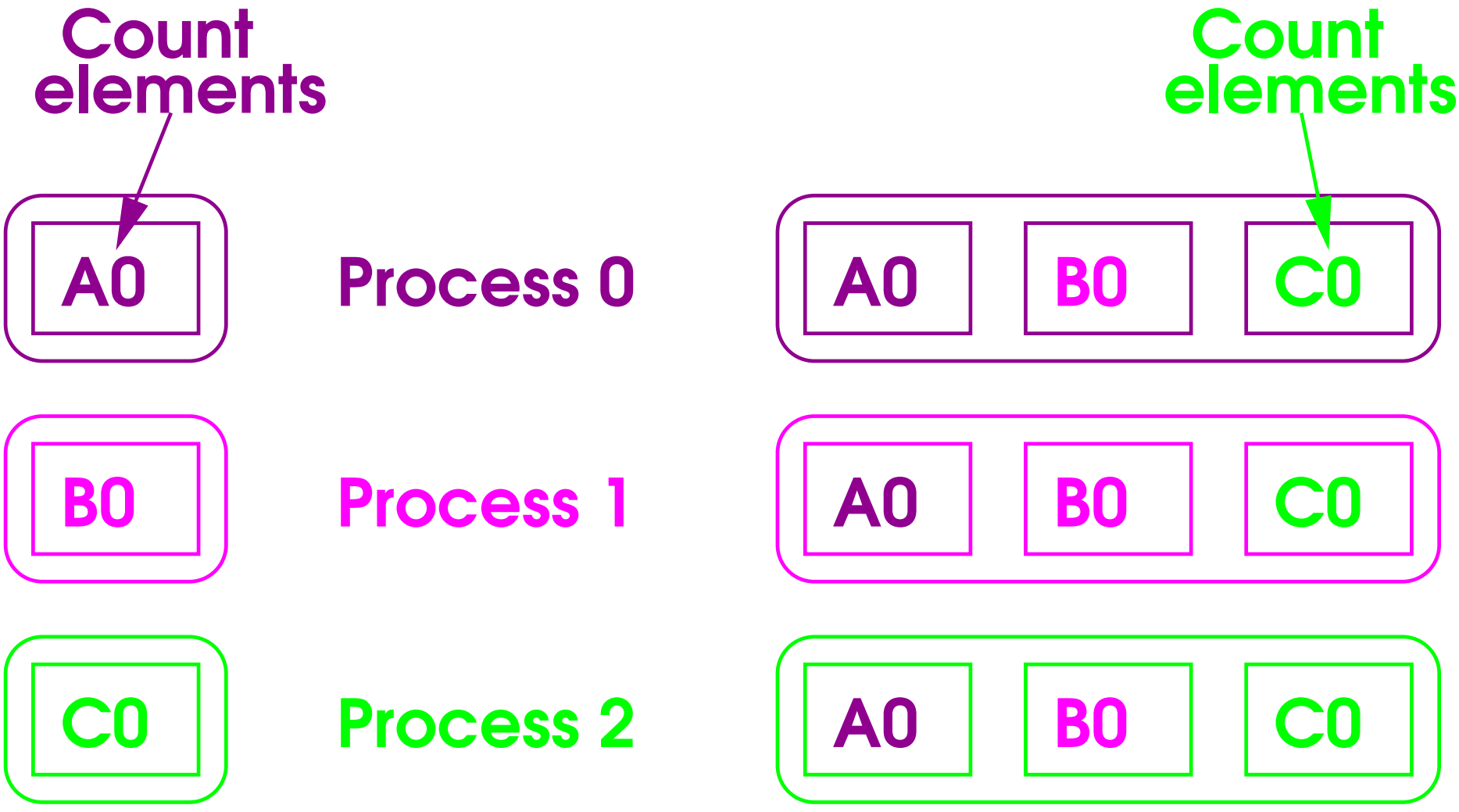
- Change MPI_Gather to MPI_Allgather

  And Gather to Allgather for C++

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 )
```

C++ example:

```
double sendbuf [ 100 ] , recvbuf [ 30 ] [ 100 ] ;
MPI::COMM_WORLD . Allgather (
        sendbuf , 100 , MPI::DOUBLE ,
        recvbuf , 100 , MPI::DOUBLE )
```

Alltoall

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

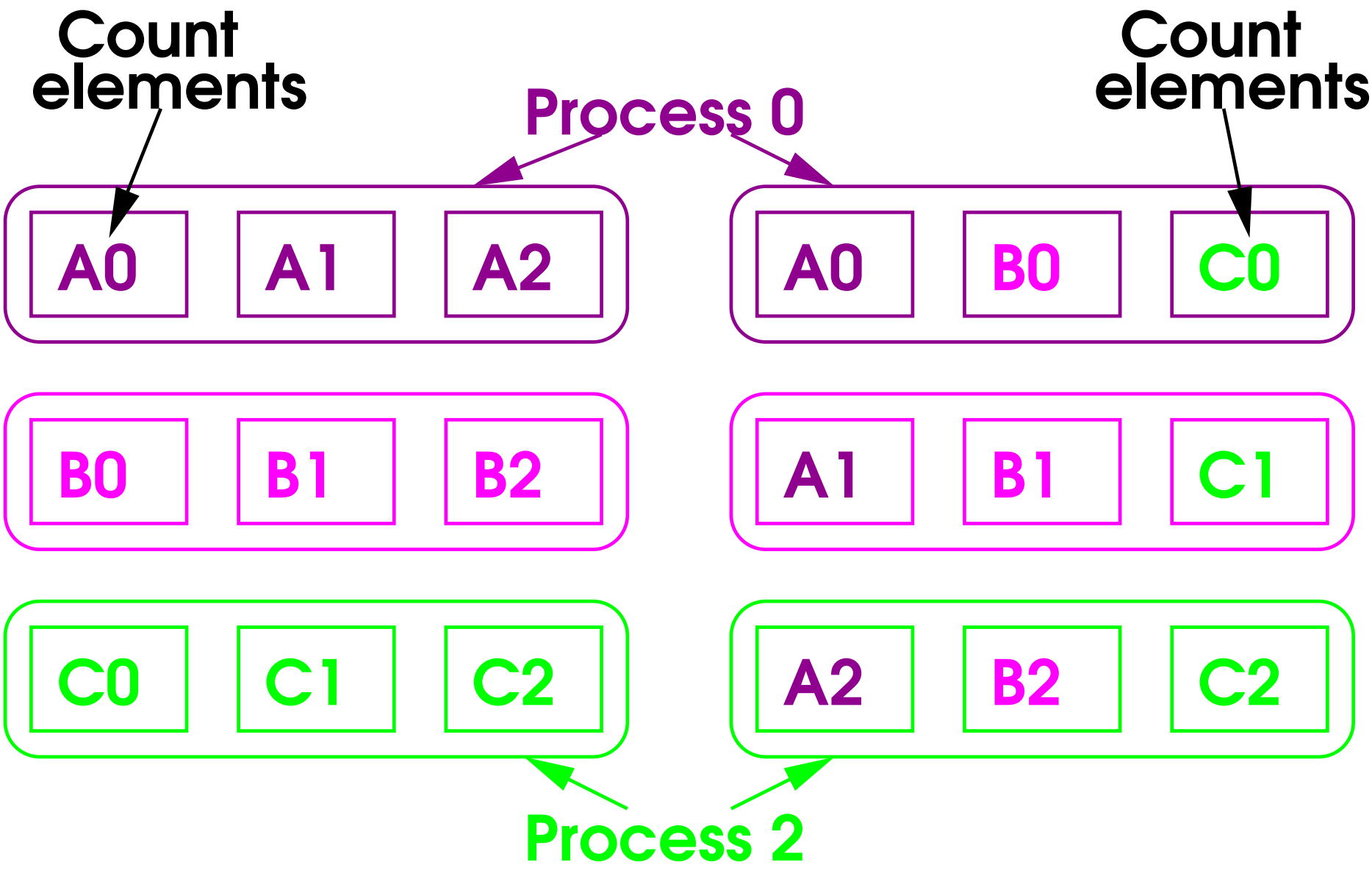
- Just change MPI_Allgather to MPI_Alltoall
  And Allgather to Alltoall for C++

- 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

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
int process_Rank, size_Of_Cluster;

 MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

 printf("Hello World from process %d of %d\n", process_Rank, size_Of_Cluster);

 MPI_Finalize();
return 0;
}
```

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- Schedule your for and sections
- Memory model
- OpenMP 3.0 and Tasks

SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct …. Coming in OpenMP 3.0

Discussed later

The loop worksharing Constructs

- **The loop workharing construct splits up loop iterations among the threads in a team**

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```

Loop construct name:

- **C/C++: for**
- **Fortran: do**

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop worksharing Constructs
A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

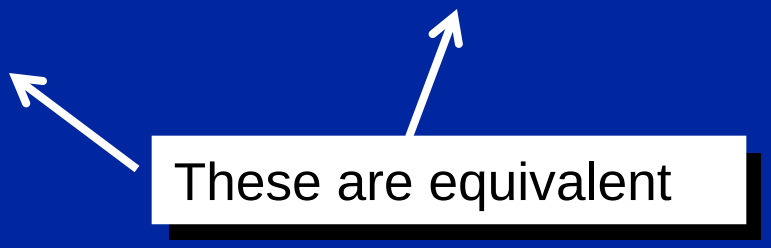
```
#pragma omp parallel
#pragma omp for
        for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
```



Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index "i" is private by default

Remove loop carried dependence

```
int i,  A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    int j = 5 + 2*i;
    A[i] = big(j);
}
```

Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];    int i;
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").
  - Compiler finds standard reduction expressions containing "op" and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in "list" must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN*	Largest pos. number
MAX*	Most neg. number

Exercise 4

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number changes made to the serial program.

Using CodeBlocks for OpenMP

Steps to install CodeBlocks:

1. Install new version of CodeBlocks.

(Available at http://sourceforge.net/projects/codeblocks/ Click on "Download codeblocks-13.12mingw-setup-TDM-GCC-481")

2. Install MinGW (Available at http://sourceforge.net/projects/mingw/files/Installer/ Click on "mingw-get-setup.exe"). While installing, select "mingw32-gcc-g++" and click "Apply Changes" under Installation tab.

3. Replace "C:\Program Files (x86)\CodeBlocks\MinGW" folder with "C:\MinGW" folder.

4. To create new project in CodeBlocks

File > New > Project > Console Application > Select C/C++ > Give Project title and folder path > Next > Compiler : GNU GCC Compiler > Finish

5. Sample Code for Main.c

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
#pragma omp parallel
{
printf("Hello World from thread = %d of %d\n", omp_get_thread_num(),
omp_get_num_threads());
}
return 0;
}
```

6. Add compiler flag "-fopenmp"

Settings > Compiler > Compiler Settings > Other Options > type ***-fopenmp*** > ok

7. Link "libgomp-1.dll"

Settings > Compiler > Linker Settings > Add > Set the path to ***C:\Program Files (x86)\CodeBlocks\MinGW\bin\libgomp-1.dll*** > Ok > Ok

8. Run the program.