Programming with MPI

Attributes and I/O

Nick Maclaren

nmm1@cam.ac.uk

May 2008

Address-Sized Integers

- MPI needs address–sized integers

Mostly for purposes we haven't come to yet
None of Fortran 90, C90 or C++ have them

- MPI's C type is called MPI_Aint

Don't use C99 intptr_t (or size_t or ptrdiff_t)

- Its Fortran 90 one is specified by:
  INTEGER(KIND=MPI_ADDRESS_KIND)

For Fortran 77, see the MPI documentation

Attributes (1)

Properties that are attached to a communicator

The standard ones aren't very useful, unfortunately
They are all effectively integer values

But here they are, for information:

MPI_TAG_UB – upper bound for tag value
a value from 32767 to MAXINT
Most people simply use the range 0...32767

Attributes (2)

MPI_HOST – host process rank, if any
MPI_PROC_NULL if there isn't one
Its meaning is defined by the implementation

MPI_WTIME_IS_GLOBAL Boolean value
True if clocks are synchronised
We will discuss this in more detail shortly

MPI_IO – rank of a node that can do I/O
We will discuss this in more detail shortly

Attributes (3)

Implementations may define other attributes
See their documentation for which ones, if any

You can also define your own – mentioned later
Attributes can do a lot more than covered here

MPI 2 added extra functionality, too

Reading Attributes (1)

- The specification is a considerable mess

The only area of MPI where that seems to be so

We need only the read attribute function

MPI_Comm_get_attr (new name)

MPI_Attr_get (old name)

- Their specifications are slightly different

MPI_Attr_get returns different types

Examples only for MPI_Comm_get_attr

Reading Attributes (2)

Returns a Boolean flag saying if attr. is set
But a missing attribute is also an error!
I have no idea why the flag exists at all

Implementations have added to the confusion
Errors are not always fatal when they should be

- Safe rule is to test both for success

Use result only if no error AND flag is True

Reading Attributes (3)

Reading standard attributes should always work
but don't trust implementors here

It's best to use the general code even for them

- Following examples provide the mumbo jumbo

Let's use MPI_TAG_UB as an example

It is one of the simplest built–in ones

Fortran and Attributes (1)

- You specify the result variable directly

Always INTEGER(KIND=MPI_ADDRESS_KIND)

MPI_WTIME_IS_GLOBAL is kludged up

0 means False; 1 means True;

MPI_Attr_get usually returns plain INTEGER

Except MPI_WTIME_IS_GLOBAL is LOGICAL

Fortran and Attributes (2)

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: maxtag
INTEGER :: error
LOGICAL :: flag

CALL MPI_Comm_get_attr ( MPI_COMM_WORLD ,
      MPI_TAG_UB , maxtag , flag , error )
IF ( error /= MPI_SUCCESS .OR. .NOT. flag )      &
      maxtag = 32767
```

C and Attributes (1)

- All results are returned as pointers

The argument is a pointer to a pointer
Its type is void * for arcane C reasons

- You are interested in the value pointed to

The type of its value is always MPI_Aint

MPI_Attr_get returns plain int

C and Attributes (2)

```
MPI_Aint minmax = 32767 , * maxtag ;
int flag , error ;

error = MPI_Comm_get_attr ( MPI_COMM_WORLD ,
        MPI_TAG_UB , & maxtag , & flag ) ;
if ( error != MPI_SUCCESS || ! flag )
        maxtag = & minmax ;
```

Timer Synchronisation (1)

This means synchronisation across processes
I.e. are all results from MPI_Wtime consistent?

Almost always the case on SMP systems
Will often be the case even on clusters

- Generally, try to avoid assuming or needing it

Rarely compare timestamps across processes

- If you use only local intervals, you are OK

Time passes at the same rate on all processes

Timer Synchronisation (2)

Beyond that is a job for real experts only

Parallel time is like relativistic time
Event ordering depends on the observer

There is a solution in directory Posixtime
Functions to return globally consistent time

I wrote this for a system with inconsistent clocks
Please ask about synchronisation if you need to

MPI and Normal I/O (1)

This means language, POSIX and Microsoft I/O

There are serious problems – not because of MPI
Caused by the system environment it runs under

- At one extreme, no normal I/O is possible

Any reasonable administrator tries to avoid that
Such systems are very rare – I don't know any

- This course will not cover such systems

Read the implementation documentation
Or ask your administrator to help you

MPI and Normal I/O (2)

- At the other extreme, all processes can do I/O

And even share a filing system (e.g. via NFS)
Most administrators set up systems like that

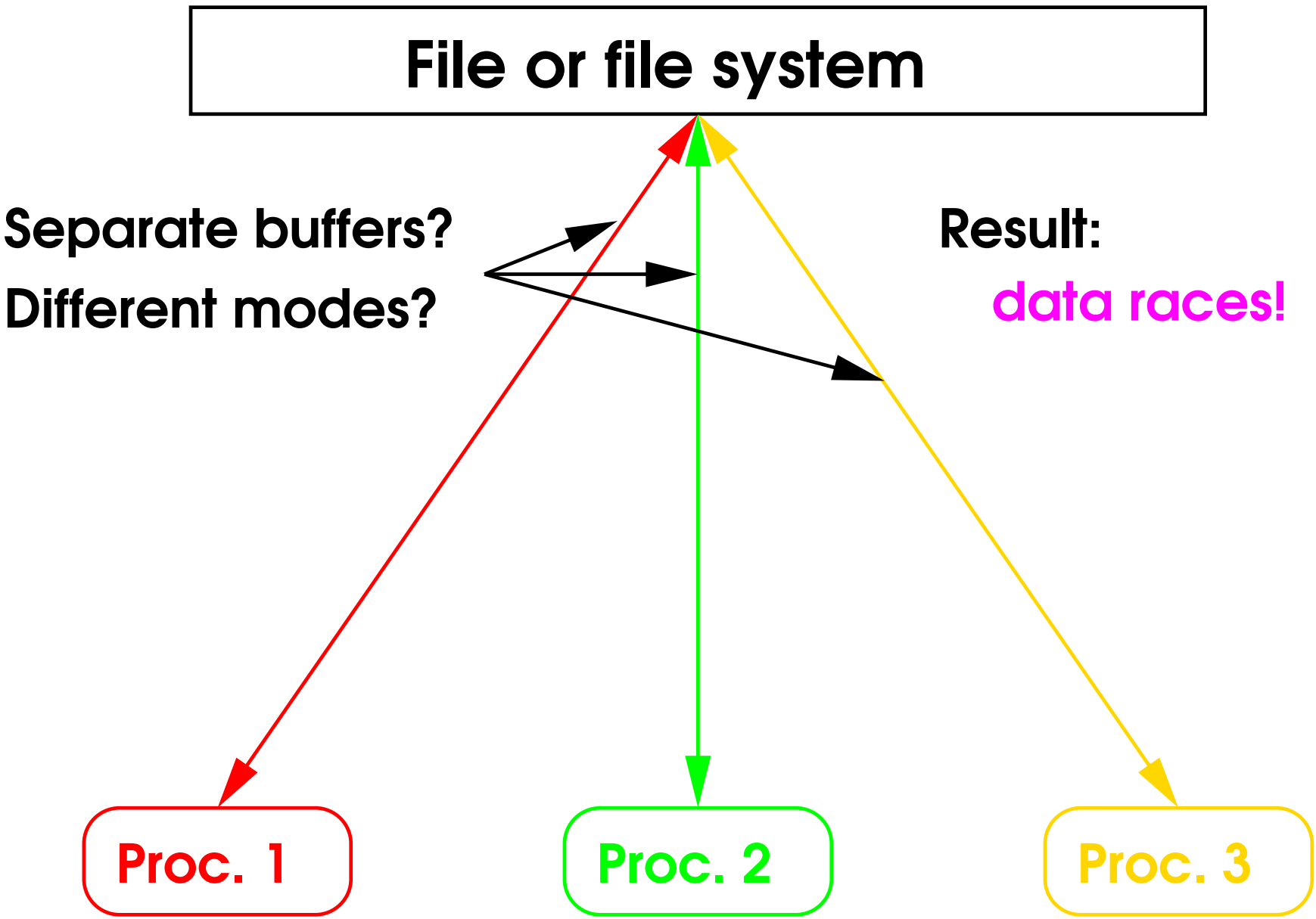
- One intermediate position is fairly common

Only one process can do normal I/O
That process is usually (almost always?) zero
in MPI_COMM_WORLD, of course

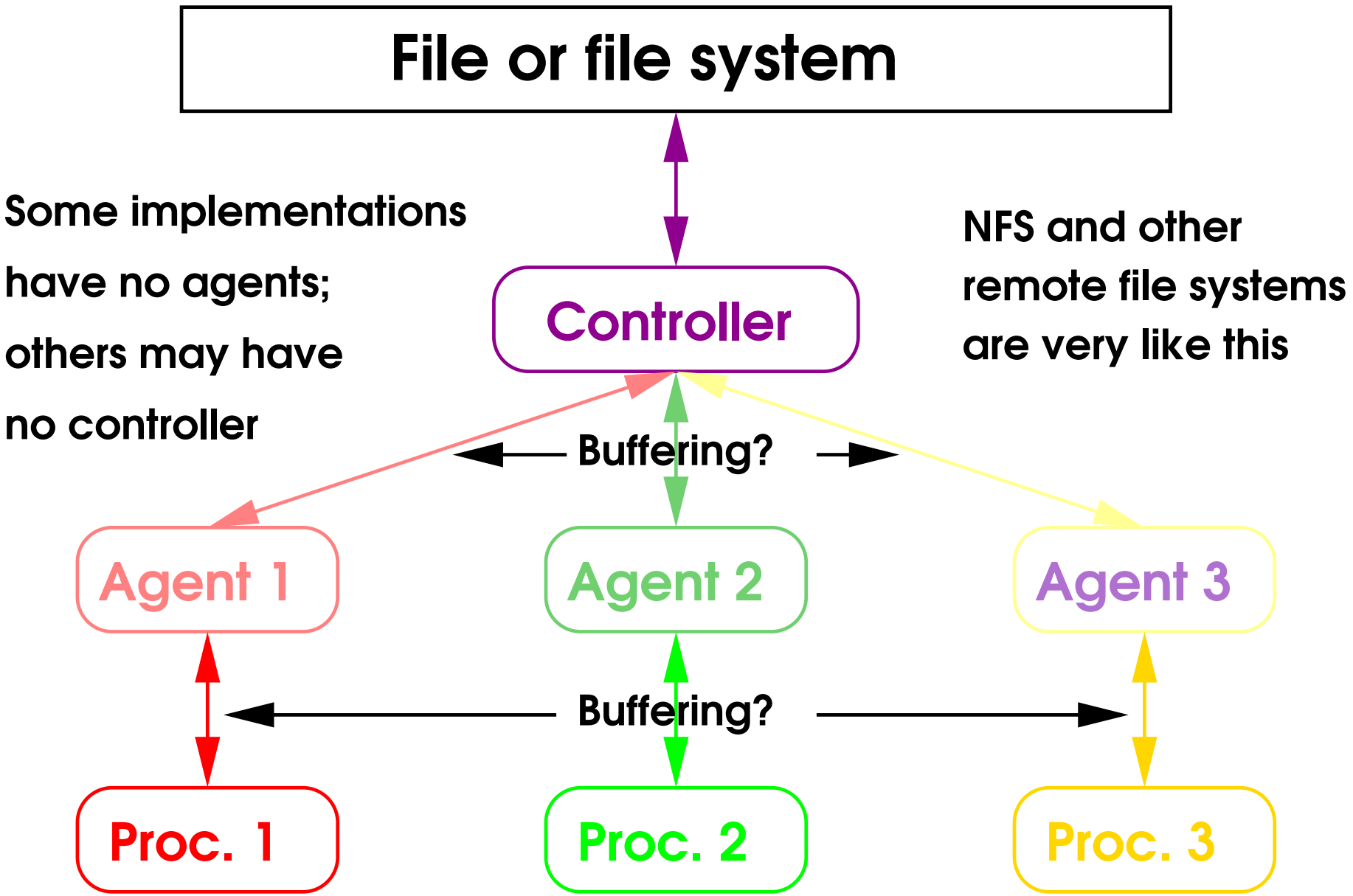
- This course will cover both types of system

And warn you about possible problems with them

Shared I/O Descriptors



Agent–based I/O Handling



MPI and Normal I/O (3)

There are two, very different, classes of file

- Normal named and scratch files
- stdin, stdout and stderr

Differences are caused by the system environment
E.g. clusters of distributed memory systems
Or shared file descriptors on SMP systems

- These issues are NOT specific to MPI

Other parallel interfaces have the same problems

MPI_IO Attribute (1)

- One attribute says if normal I/O is possible
  MPI_IO attached to MPI_COMM_WORLD

- Unfortunately, it isn't very useful in practice

It doesn't distinguish the two classes of I/O
It doesn't say if you have a shared filing system

I don't always use it, though I should
It is important if only one process can do I/O
So you need it for maximum portability

MPI_IO Attribute (2)

The value can be any of the following:

- MPI_ANY_SOURCE
  All processes in the communicator can do I/O
- The number of the local process
  This process can do I/O, but not all can
- Another process number
  This process can't, but that numbered one can
- MPI_PROC_NULL
  No process in the communicator can do I/O

We now ask "Can this process do I/O?"

Fortran Local I/O Test

```
LOGICAL :: Local_IO , flag
INTEGER(KIND=MPI_ADDRESS_KIND) :: result
INTEGER :: myrank , error

CALL MPI_Comm_rank (        &
       MPI_COMM_WORLD , myrank , error )
CALL MPI_Comm_get_attr (        &
    MPI_COMM_SELF , MPI_IO , result , flag , error )
Local_IO = ( error == MPI_SUCCESS .AND.        &
       flag .AND.        &
       ( result == myrank .OR.        &
              result == MPI_ANY_SOURCE )
```

C Local I/O Test

```
int Local_IO , flag , myrank , error ;
MPI_Aint * result ;

MPI_Comm_rank ( MPI_COMM_WORLD ,
      & myrank ) ;
error = MPI_Comm_get_attr ( MPI_COMM_SELF ,
      MPI_IO , & result , & flag ) ;
Local_IO = ( error == MPI_SUCCESS && flag &&
      ( * result == myrank .OR.
            * result == MPI_ANY_SOURCE ) ) ;
```

Unfortunately . . .

OpenMPI misimplements MPI_IO, badly

It defines the name, but doesn't set a value

So the variable flag is set to false

MPICH does better, here, but has other bugs

Most proprietary MPIs do better, too

This means that you can't test the first example

Shared File Access (1)

- For now, assume all processes can do I/O

- Assume all processes share a filing system

Directly, using POSIX, or indirectly, using NFS
Or with the Microsoft and other equivalents

- Here are some rules on how to use files safely

Shared File Access (2)

- Always use write–once or read–many

That applies to the whole duration of the run

- All updates and accesses must be considered

Including any that are done outside MPI

I.e. if a file is updated at any time in the run
only one process opens it in the whole run

Any number of processes may read a single file
provided that no process updates it

Directories (1)

- Regard a directory as a single file (it is)

If you change it in any way in any process

- Don't access it from any other process

Creating a file in it counts as a change, of course

If you do, a parallel directory listing may fall over!
Listing a read–only directory is safe

- Can create and delete separate files fairly safely

[But not under Microsoft DFS, I am afraid]
Create and delete any single file in one process

Directories (2)

You can do a bit better, fairly reliably
[But not under Microsoft DFS, I am afraid]

- Close all shareable files in all processes

Including all output files in shareable directories

- Call MPI_Barrier on MPI_COMM_WORLD

Wait (call sleep) for 5 seconds or so

- Call MPI_Barrier on MPI_COMM_WORLD

If that still doesn't synchronise the filesystem

- Increase the time or consult an expert

Apologia

This all sounds ridiculous, but I am afraid it isn't
Synchronisation in shared file systems is chaos
Whether NFS, Microsoft DFS, Lustre or other

- Directory access is a particularly unreliable area

POSIX has no directory synchronisation
NFSv3 has race conditions on directories
Microsoft DFS doesn't support parallel use
Lustre etc. are too complicated to describe

And so on ....

Working Directory (1)

Most clusters are set up conveniently

- Then, all processes share a working directory

With luck, that's controllable or your home directory
The details are very system–dependent, as usual

- PWF/MCS Condor is (was?) not one such

Working Directory (2)

I had better mention unfriendly system setups

Each process may have separate working directory
Or there may be one, in some inconvenient location
e.g. inaccessible from the development one

Need way to copy source data files into them
And to copy any result files out of them

- All that is outside the scope of this course

Contact your administrator for help

Scratch Files

Don't assume where scratch files go
That statement applies even on serial systems
It is even more complicated on parallel ones

It's common to have shared working directories
But separate, distributed scratch directories

- Just a warning – clean code rarely has trouble

Standard Units (1)

Issues arise from implementation details

- Almost always show up with output

Probably just because almost all programs use it!

- It is an almost unbelievable can of worms

Don't even try to program round the problems
Only solution is to bypass the issue entirely

- These issues are NOT specific to MPI

Other parallel interfaces have the same problems

Standard Units (2)

What can happen with standard input (i.e. stdin)?

- The stdin of mpiexec may be ignored

All input to the processes is empty (i.e. /dev/null)
Or the file descriptors may not even be provided

- The data may be copied (“spooled”)

Each process gets its own copy of the lot

- File descriptors may be shared (i.e. with dup)

Data are consumed in units of buffer sizes
The only sequencing is first come, first served

Standard Units (3)

Plus hybrid approaches, variants and so on

Also, I haven't seen everything

- Output is similar, but in reverse

Fairly rare for it to be simply thrown away

That applies to both stdout and stderr

Shared Descriptors

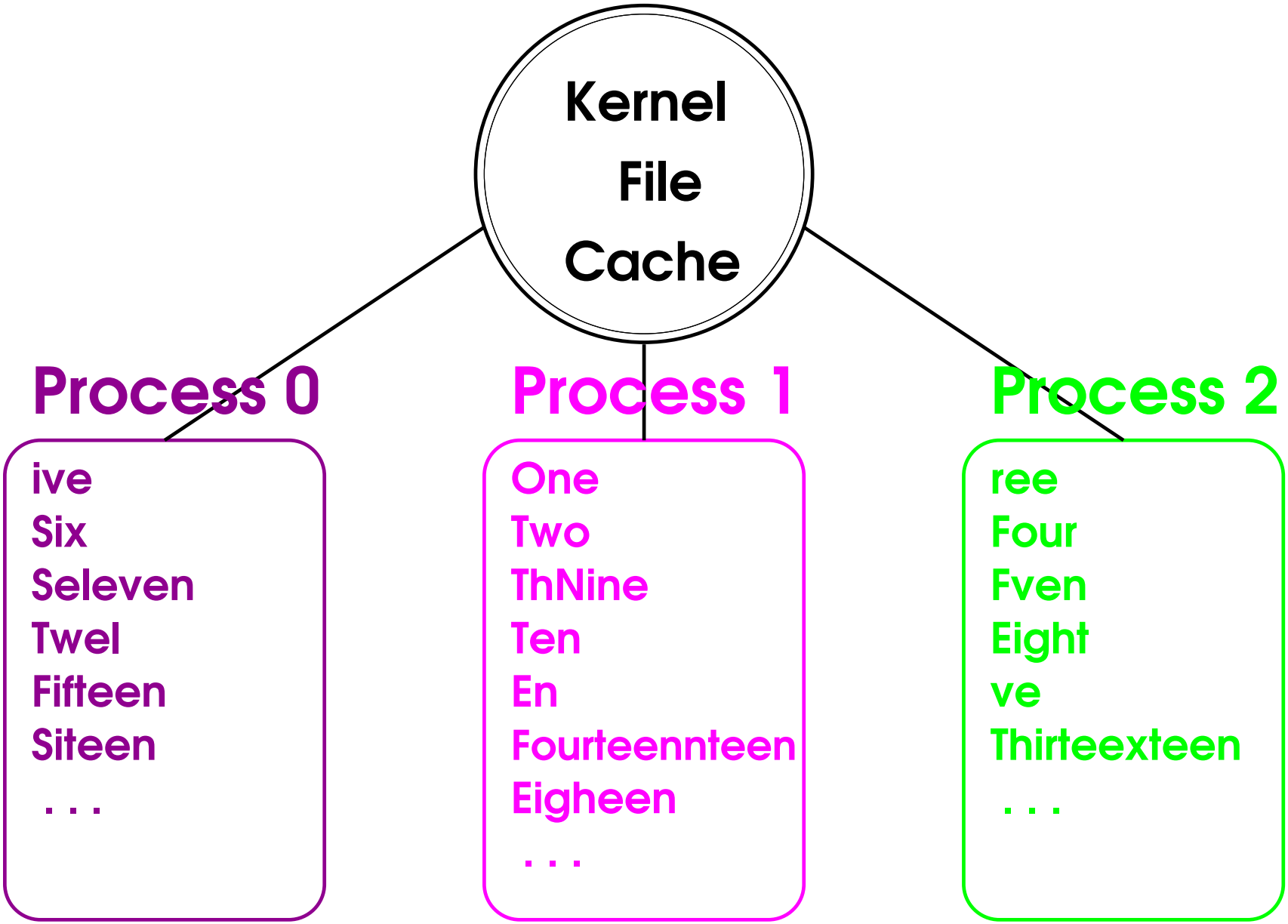
Use input from this stream as an example:

One
Two
Three
Four
Five
. . .

And buffers of size 10 for clarity

Typical values are 512, 4096 and more

Shared Descriptor I/O



Separate Systems

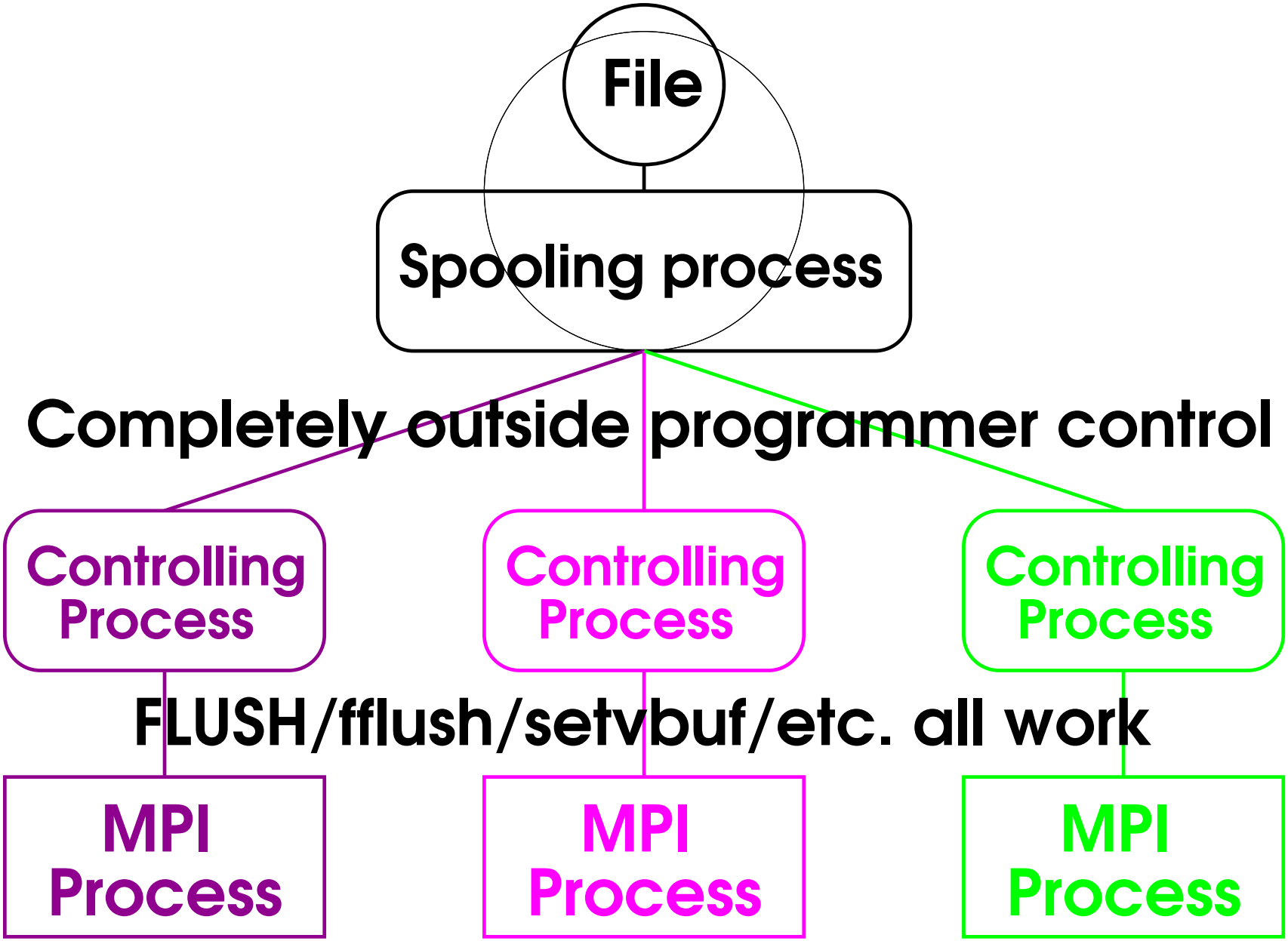
Can also happen when using separate systems
Usually when running under a job scheduler

I have been asked why this is, many times

Often, the mpiexec command does the actual I/O
So how do the processes talk to that?
Often indirectly, when under job schedulers

Here is a graphical explanation

Typical Job Scheduler I/O



Avoiding the Mess

The "right" solution is also the simplest
Only one process does stdin/stdout I/O

It does all the reading from stdin
It broadcasts or scatters it to the others

It gathers all of the output from the others
And then it writes it to stdout

It can also order I/O by process number
And can also be done for file I/O

Finding an I/O Process (1)

This is left as an exercise for the student!
Have already learnt all of the techniques needed

First step is find a suitable I/O process
This should be implemented as a collective

Each process checks whether it can do I/O
Then use reduction to find the lowest rank

We have covered all of the features needed

Finding an I/O Process (2)

Write a function to do the following:

- Check for local I/O as shown above
  - If so, store the local rank
  - If not, store a large number
- Use MPI_Allreduce and MPI_MIN
  - If the result is a rank, we are OK
  - If not, no process can do I/O

Return the I/O process rank or abort

Missing MPI Feature

There is no MPI datatype for MPI_Aint
or Fortran KIND=MPI_ADDRESS_KIND

- You therefore can't transfer the results directly

Convert attributes to plain integers first

Tedious, but no more than that – it's rarely needed
You don't need to do it for the above procedure

There is a way round it for Fortran only
It's not pretty and you may not guess it . . .

Handling Standard I/O (1)

The simplest case is when all processes match
All do I/O together, all of the same length
You can pad messages/lines to a fixed length

Use MPI_Bcast, MPI_Scatter and MPI_Gather

It's only a little trickier when the lengths vary
You need MPI_Scatterv and MPI_Gatherv

- You have already done almost all of that!

Handling Standard I/O (2)

Or you can use point–to–point in several ways
Emulating the collectives is very easy
But the real purpose is to do something they don't

You can send a series of messages point–to–point
And terminate the series with a special message

- This is still programming collectively

All processes are doing I/O, or none are
In between I/O, all can get on with calculation

- It is fairly easy to avoid deadlock

Non-Collective I/O (1)

Beyond that, it gets hairier, rapidly

Simplest case is when both processes expect it
Isn't easy to arrange, if both do calculation

- Worst case is when I/O can occur at any time

It can be done, but it's a foully complicated task

- If you need that, simplify the problem

The I/O process does no normal calculation
It does just I/O and other forms of control

Non-Collective I/O (2)

- I am NOT being patronising!

I needed some flexible I/O to stdout
I wrote some simple non–collective code
I fixed bugs one and two quite easily
But three was a bug in my design

I then redesigned the code to be collective
Still using point–to–point calls
That worked, almost the first compile

The Master/Worker Model

Think of the master/worker model
The master divides the problem into tasks
And assigns tasks to the workers

- The master has all of the control logic
- The workers do all of the calculation

A process can be both a master and worker
- That is very tricky to get right

Most people code deadlocks when trying to do it

I did . . .

Error Messages etc.

- You can just write to stderr or equivalent

Fortran users may need to use FLUSH

It may well get mangled (reasons given above)
It may get lost on a crash or MPI_Abort
But it's simple, and errors are rare, right?

Same applies to stdout, with some programs

- Beyond that, use a dedicated I/O process

Just as we described for stdout above

Asynchronous I/O (1)

Writing output to stdout/stderr asynchronously
While the I/O process is doing normal calculation

How to do this only if you really must ...

- Attach a suitable buffer in each process

Enough for the process's total output

- Use buffered sends from all processes

At last, send a special "end of transmission"

Asynchronous I/O (2)

- Use a distinctive tag for all messages

Or a separate copy of MPI_COMM_WORLD

- Whenever convenient in the I/O process:

Use MPI_Iprobe looking for that tag

And then transfer any messages to the output

- Flag a process as dead after "EOT"

- Before the I/O process shuts down

Loop until all other processes are marked dead

Do this by waiting on the tag

Asynchronous I/O (3)

Common failure modes of that approach:

- If another process dies before terminating

The I/O process will wait forever

- If the I/O process dies

You lose all remaining output anyway

TANSTAAFL