Programming with MPI

One-sided Communication

Nick Maclaren

nmm1@cam.ac.uk

October 2010

What Is It?

This corresponds to what is often called RDMA
That is “Remote Direct Memory Access”
[It is called RMA in MPI]
Much loved by Cray and the DoD/ASCI people

One process accesses the data of another
Potentially more efficient on SMP systems
It was added in MPI 2, but is not often used

- I will explain why I don’t like RDMA

Then describe how it can be used semi–safely

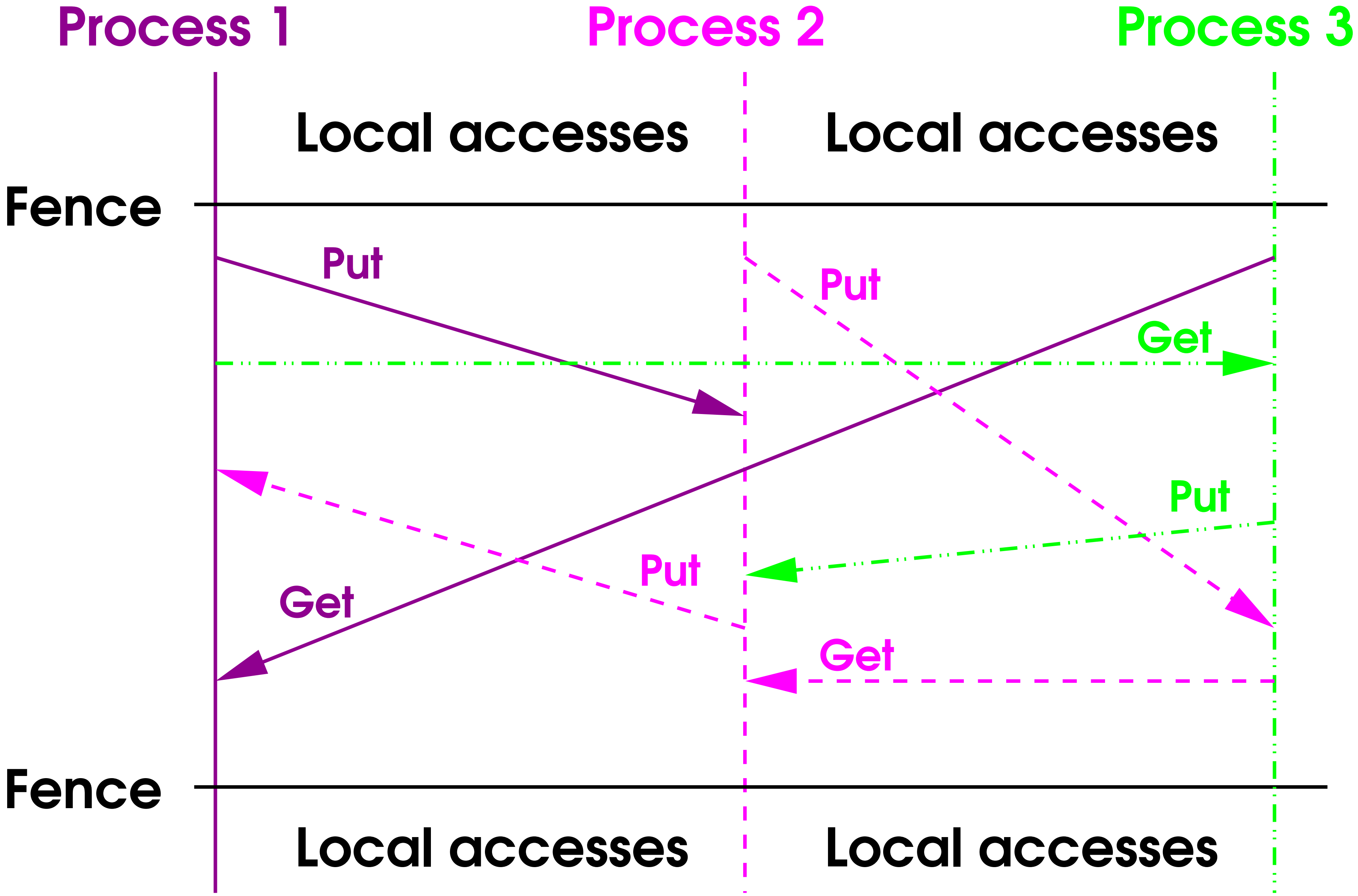
RDMA Models

- Active RDMA is batching up sends and receives
A bit like non–blocking using MPI_Waitall
Not too hard to use correctly, but needs discipline

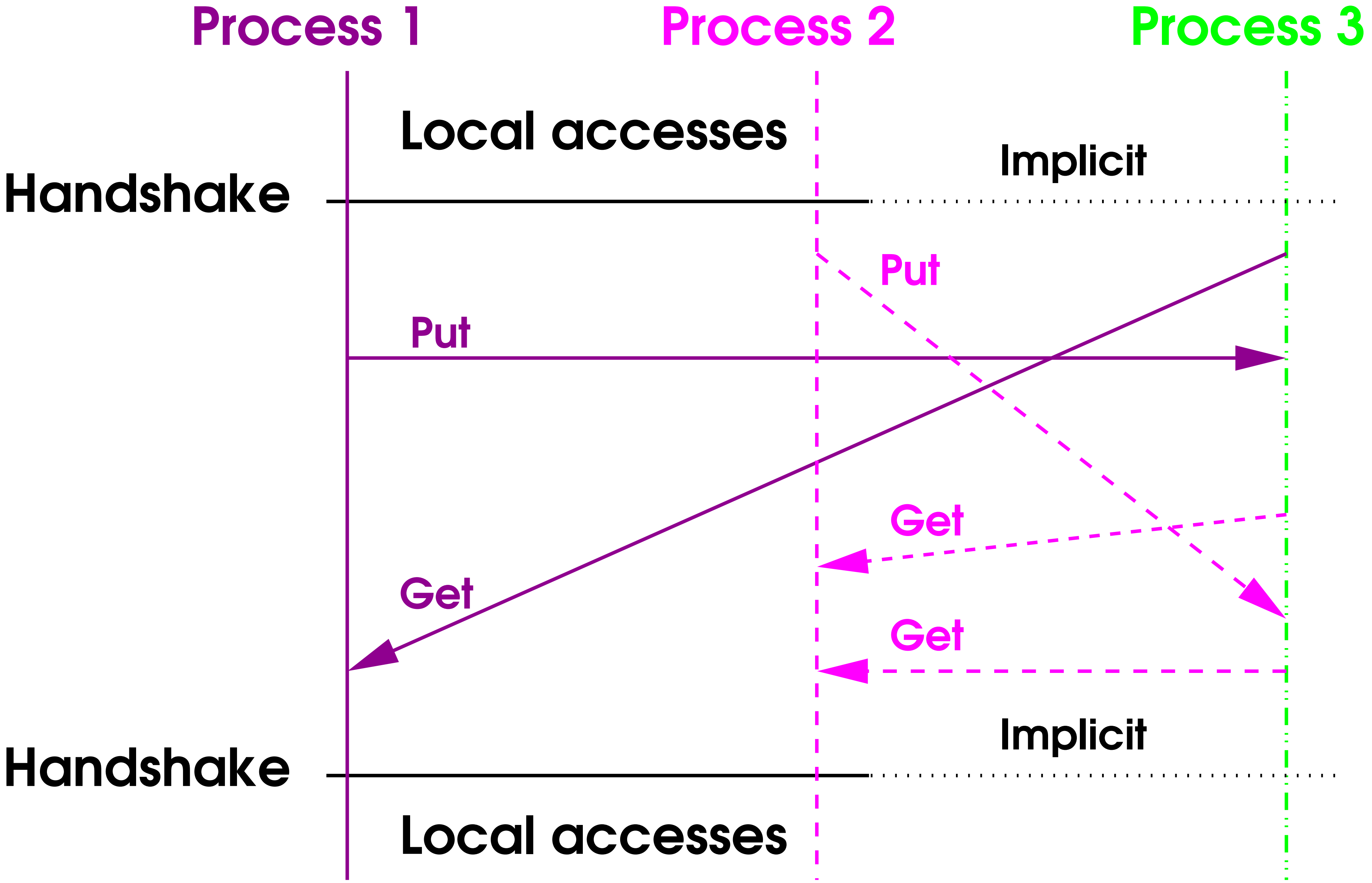
- Passive uses uninvolved process’s memory
Much harder to use and less portable
PGAS languages effectively use this model

- Lastly, can use true virtual shared memory
Using all processes’ memory as if it were local
MPI does not support this, for very good reasons

Active (Fenced) Communication



Passive Communication



Problems with RDMA (1)

RDMA is often said to be easier to use correctly

Twaddle!

How does other process know its data is in use?
Get it wrong, and almost unfindable chaos ensues

- Adding handshaking often makes it pointless

Most specifications don’t say how to handle that
MPI does, which makes it fairly complicated

Problems with RDMA (2)

Another problem is that it handicaps analysis tools
With two–sided, can diagnose unmatched transfers
But there's no concept of matching for one–sided

Its semantics are similar to SMP threading

- Lots of lovely potential race conditions
- And no realistic tools to detect them

Similar remarks apply to non–trivial tuning

Those problems are insoluble – provably so
Most people's experiences are very negative

Problems with RDMA (3)

Next problem is implementation and progress
Passive RDMA needs a remote agent on memory

May hang until far process reaches an MPI call
With passive use, that may never happen
Can also occur with active, but less likely

- Deadlock may happen even in correct code

MPI forbids that for two–sided and (mostly) active

Some subtle problems even on SMP systems

Problems with RDMA (4)

Last problem is performance complications

- On commodity clusters, it may well be slower

Remote agent may interfere with MPI process

- Probably faster on specialist HPC clusters

Cray etc. have hardware and kernel RDMA
Transfers data without any action by MPI process

- On SMP systems, it's very hard to guess

+ May well have lower communication overhead
– But may cause serious cache and TLB problems

Most Likely Benefit

- One logical transfer is many actual ones

I.e. you need to transfer lots of objects together
Essentially an alternative to packing up into one array

- On SMP system or specialist HPC cluster

Does NOT gain you anything using Ethernet
Nor Infiniband with commodity software

TCP/IP and Ethernet need a kernel thread
Currently, so does the OpenIB Infiniband driver

A SMP System Myth

The following is NOT true, in general:
One–sided transfers use fewer memory copies

True in theory, but generally not in practice
Probably true only for MPI_Alloc_mem memory

- For all Unix–like systems, including Microsoft’s
Process A cannot access process B’s memory

- Can only if using a shared memory segment
And almost all data areas aren’t in one of them
⇒ Two memory copies needed for it, as well

Recommendation

- Stick to simple forms of active communication

Only one taught here is collective–like transfers
Group pairwise transfers are also described, roughly

- Be extremely cautious about deadlock

Use only methods that don't rely on progress details

- Scrupulously adhere to MPI's restrictions

Designed to maximise reliable implementability

That is, naturally, all that this course teaches

Preliminaries

Recommended to create info object, for efficiency
Most of its uses are for other MPI 2 facilities
Do this on each process, but it is not collective

- Asserts that won't use passive communication

Curiously, no option to specify a read–only window

Also a relevant function MPI_Alloc_mem
May help with performance on some systems
But is NOT recommended for use in Fortran
It's fine in MPI 3.0, which uses Fortran 2003

Fortran Example

```
INTEGER :: info , error

CALL MPI_Info_create ( info , error )
CALL MPI_Info_set ( info , "no_locks" ,        &
        "true" , error )

!  Use the info object

CALL MPI_Info_free ( info , error )
```

C Example

```
MPI_Info info ;
int error ;

error = MPI_Info_create ( & info ) ;
error = MPI_Info_set ( info , "no_locks" , "true" ) ;

/* Use the info object */

error = MPI_Info_free ( info ) ;
```

Windows

A window is an array used for one–sided transfers

- Must register and release such arrays collectively

Args are address, size in bytes and element size

- All of them can be different on each process

Same remark applies to the element type of the array

- Offsets are specified in units of element size

Like array indices into the target window

- Size of zero means no local memory accessible

Alignment

- Strongly advise to align correctly for datatype

Only a performance issue, but may be significant

Some systems may benefit from stronger alignment
Read the implementation notes to find that out

- Element size need not match the element type

But simplest use is to make it match

It can be 1 if using byte offsets

- In that case, it’s your job to get it right

Fortran Example

```
 REAL ( KIND = KIND ( 0.0D0 ) ) :: array ( 1000 )
 INTEGER :: info , window, size, error
INTEGER ( KIND = MPI_ADDRESS_KIND ) :: win_size

 ! Create the info object as described above
 CALL MPI_Sizeof ( array , size , error )
 win_size = 1000 * size
 CALL MPI_Win_create ( array , win_size ,        &
         size , info , MPI_COMM_WORLD ,        &
         window , error )

 CALL MPI_Win_free ( window , error )
```

C Example

```
double array [ 1000 ] ;
MPI_Info info ;
MPI_Win window ;
int error ;
MPI_Aint win_size ;

/* Create the info object as described above */
win_size = 1000 * sizeof ( double ) ;
error = MPI_Win_create ( array , win_size ,
        sizeof ( double ) , info , MPI_COMM_WORLD ,
        & window ) ;

error = MPI_Win_free ( window ) ;
```

Horrible Gotcha

- The window must be an ordinary data array

Not an MPI standard issue, and is not stated there
It's a hardware, system and compiler one
Applies to other RDMA and asynchronous use, too

- Do not use Fortran PARAMETER arrays
- Do not use C/C++ static const arrays
- Or anything any library returns that might be
- Or anything else that might be exceptional

At best, the performance might be dire

Fencing

Exactly like MPI_Barrier, but on a window set

Assertions described shortly – ignore for now

Fortran example:

```
CALL MPI_Win_fence ( assertions , window , error )
```

C example:

```
error = MPI_Win_fence ( assertions , window )
```

C example:

```
window.Fence ( assertions )
```

Use of Windows (1)

Rules for use of windows apply much more generally
But are easiest to describe in terms of fencing

Fences divide time into sequence of access epochs
Each window should be considered separately

⇒ So consider one epoch on one window

No restrictions if not accessed remotely in epoch

- It includes local writes from it using RDMA

That is unlike the rules for MPI_Send

Use of Windows (2)

Window is exposed if it may be accessed remotely
If the window is exposed:

- Mustn't update any part of window locally

Seems unnecessary, but are good reasons for this
Can use RDMA to local process to bypass this

- Any location may be updated only once

And not at all if it is read locally or remotely
The standard write–once–or–read–many model

Accumulations are an exception – see later

Optimisation Issues

Bends language standards in several subtle ways

- Can cause accesses to get out of order

Very rare for C and C++ – and truly evil if it does

- Almost all such problems are user error

Fortran is more optimisable, and it can happen

- Window should have ASYNCHRONOUS attribute

So should any dummy argument that it is passed to during any access epoch

- Simplest not to pass it during an access epoch

Assertions (1)

This is an integer passed to synchronisation calls
Can be combined using logical OR (IOR or '|')
Purely optional, but may help with performance

- If you get them wrong, behaviour is undefined
Pass the argument as 0 if you are unsure

- Fall into two classes: local ones and collective
Latter are supported only by MPI_Win_fence

- Apply to an epoch between synchronisations
Can be either preceding or succeeding epoch

Assertions (2)

Local assertions:

MPI_MODE_NOSTORE – about preceding epoch

The window was not updated in any way

MPI_MODE_NOPUT – about succeeding epoch

The window will not be updated by RDMA

Collective assertions:

MPI_MODE_NOPRECEDE – about preceding epoch

No RDMA calls were made by this process

MPI_MODE_NOSUCCEED – succeeding epoch

No RDMA calls will be made by this process

Assertions (3)

- If alternating computation and communication

Do the following collectively (same on all processes)

  - . . . Do some computation
  - Fence with MPI_MODE_NOPRECEDE
  - Do some RDMA communication
  - Fence with

MPI_MODE_NOSUCCEED | MPI_MODE_NOPUT

  - Do some computation . . .

And use MPI_MODE_NOSTORE when appropriate

It does not need to be the same on all processes

Transfers

MPI_Put and MPI_Get have identical syntax
Effectively start a remote MPI_Recv or MPI_Send

- Both sets of arguments are provided by the caller

Remote ones interpreted in context of target window

- Strongly recommended to match types and counts

Remote datatype must match remote element type

Like non–blocking, so don't reuse local buffers
The next synchronisation call completes the transfer

- No guarantee that it is implemented like that

Fortran Example

```
REAL :: array_1 ( 1000 ) , array_2 ( 1000 )
INTEGER :: to = 3 , from = 2 , window, error

CALL MPI_Put ( array_1 , 1000 , MPI_REAL ,        &
        to , offset_1 , 1000 , MPI_REAL ,        &
        window , error )

CALL MPI_Get ( array_2 , 1000 , MPI_REAL ,        &
        from , offset_2 , 1000 , MPI_REAL ,        &
        window , error )
```

C Example

```
double array_1 [ 1000 ] , array_2 [ 1000 ] ;
MPI_Win window ;
int to = 3 , from = 2 , error ;

error = MPI_Put ( array_1 , 1000 , MPI_DOUBLE ,
        to , offset_1 , 1000 * size , MPI_DOUBLE ,
        window )

error = MPI_Get ( array_2 , 1000 , MPI_DOUBLE ,
        from , offset_2 , 1000 * size , MPI_DOUBLE ,
        window )
```

Reminder

- The target address is not specified directly

window describes the remote array to use

offset is in units of remote element size
from start of remote window

Accumulation (1)

- A point–to–point reduction with a remote result
Accumulation is done in an undefined order

Syntax and use is almost identical to MPI_Put

Reduction operation before the window argument
Exactly the same operations as in MPI_Reduce
Only predefined operations – not user–defined

One extra predefined operation MPI_REPLACE
Simply stores the value in target location

Accumulation (2)

- You mustn’t update or access it any other way
But you can use multiple separate locations
Each can use the same or a different operation

- Cannot use it for atomic access
Can only accumulate until next synchronisation
Partly possible, but needs features not taught here

Use only one operation for one location
MPI doesn’t require it – your sanity does

Syntax Comparison

```
CALL MPI_Put ( array , 1000 ,          &
        MPI_REAL , to , offset , 1000 ,         &
        MPI_REAL , window , error )

CALL MPI_Accumulate ( array , 1000 ,          &
        MPI_REAL , to , offset , 1000 ,         &
        MPI_REAL , MPI_SUM , window , error )
```

Note only change is addition of MPI_SUM

C and C+ have identical changes

Writing Collectives

Above is enough to write collectives using RDMA
If clean and simple, you should have no trouble

MPI specifies to never deadlock but, for sanity:

- Do not overlap use of window sets

Like communicators, serialise overlapping use

- Do not overlap with other types of transfer

Serialise collectives, point–to–point and one–sided

Group Pairwise RDMA (1)

- This is a lot more complicated

So much so, this course doesn’t cover the details

- It is also a lot more error–prone

Make a mistake, and your program may deadlock
Or it may cause data corruption, with no diagnostic

- Worse, these may happen only probabilistically

I do not recommend using this, except for experts
The following is a summary of how to use it

Group Pairwise RDMA (2)

MPI_Win_post (group , assertions , window)

This registers the window for RDMA accesses
Only the processes in group may access it
MPI specifies that it does not block

MPI_Win_wait (window)

This blocks until all RDMA has completed
All processes have called MPI_Win_complete
It cancels the registration for RDMA accesses

Group Pairwise RDMA (3)

MPI_Win_start (group , assertions , window)

This registers the window for RDMA calls
You may access only the data of processes in group
It may block, but is not required to
I.e. until all processes have called MPI_Win_post

MPI_Win_complete (window)

This cancels the registration for RDMA calls
It may block, but is not required to
Circumstances too complicated to describe here

Group Pairwise RDMA (4)

You will need some of the group calls
You need not use them collectively for this
See the lecture on Communicators etc. for them

MPI_Comm_group (comm , group)

This obtains the group for a communicator

MPI_Group_incl (group , count , ranks , newgroup)

This creates a subset group from some ranks

MPI_Group_free (group)

This frees a group after it has been used

Group Pairwise RDMA (5)

That’s essentially all of the facilities

You can test for completion using MPI_Win_test
But can’t touch the window until it completes

If you want to use group pairwise RDMA:

Read the MPI standard

Passive RDMA

Did you think that the group pairwise form is bad?
Passive RDMA is simpler but much trickier

Don’t go there

Debugging and Tuning

You're on your own