MPI One sided Communication

Chris Brady
Heather Ratcliffe

“The Angry Penguin“, used under creative commons licence
from Swantje Hess and Jannis Pohlmann.

Warwick RSE

Notes for Fortran

- Since it works with raw memory pointers, these routines use a type **MPI_Aint**
  - That is an integer large enough to store a memory address
- In Fortran, this becomes
  - INTEGER(KIND=MPI_ADDRESS_KIND)

Notes on MPI_Aint

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

- Routine to add together two MPI_Aints

```
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
```

- Routine to difference two MPI_Aints

```
int MPI_Get_address(const void *location, MPI_Aint *address)
```

- Routine to get MPI_Aint address from pointer (C) or variable (Fortran)

Notes on MPI_Aint

- Mostly people don’t bother using these
- Except MPI_Get_address in Fortran
  - Result in C is almost always same as **&** operator
- Strictly should always use them when working with MPI addresses

Parallel computation
in general

Parallel concepts

- Two parts to communication
  - Communication - Put data in place
  - Synchronisation - Know that data is in place

Shared Memory

- Communication is implicit
    - Access memory directly
    - Load and store
- Synchronisation is explicit
    - Mutex objects (pthreads)
    - OpenMP CRITICAL sections (among others)

MPI - Conventional

- Communication is explicit
  - Sends and receives
- Synchronization is both
  - Implicit - Blocking operations
  - Explicit - Non-blocking operations

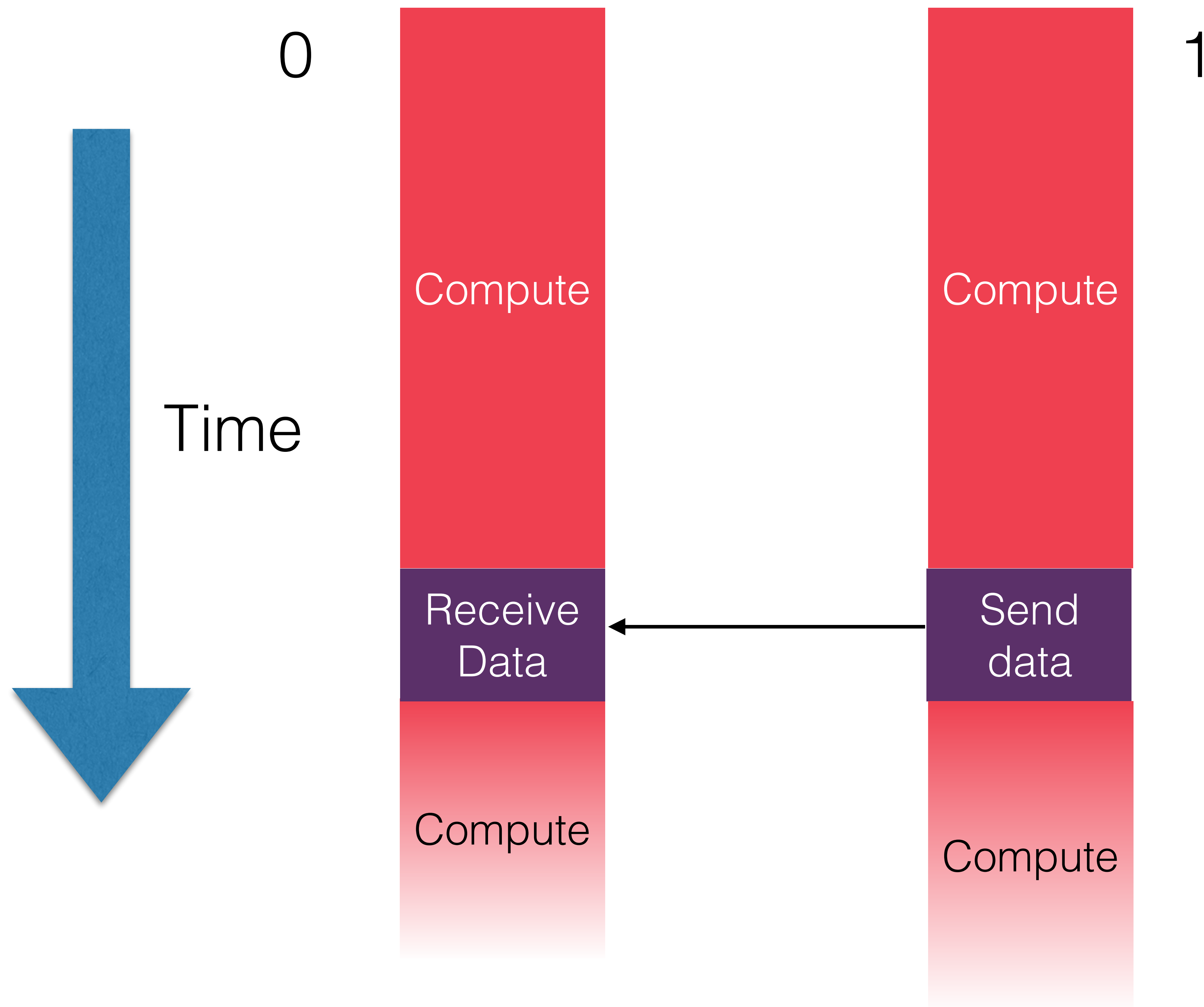
MPI - One sided

- Communication is explicit
    - Get and Put operators
- Synchronisation is explicit
    - Communication epochs
    - Guarantee no communication before epoch starts
    - Guarantee all data in place when epoch ends

Why want explicit synchronisation?

- For most purposes implicit synchronisation is fine
- Data can only be sent once it's available
- If load is well balanced data is available on all processors at the same time
- You cannot continue the next iteration until data is both sent and synchronised
    - You can sometimes split computation up so this isn't quite true
- Combining communication and synchronisation makes sense

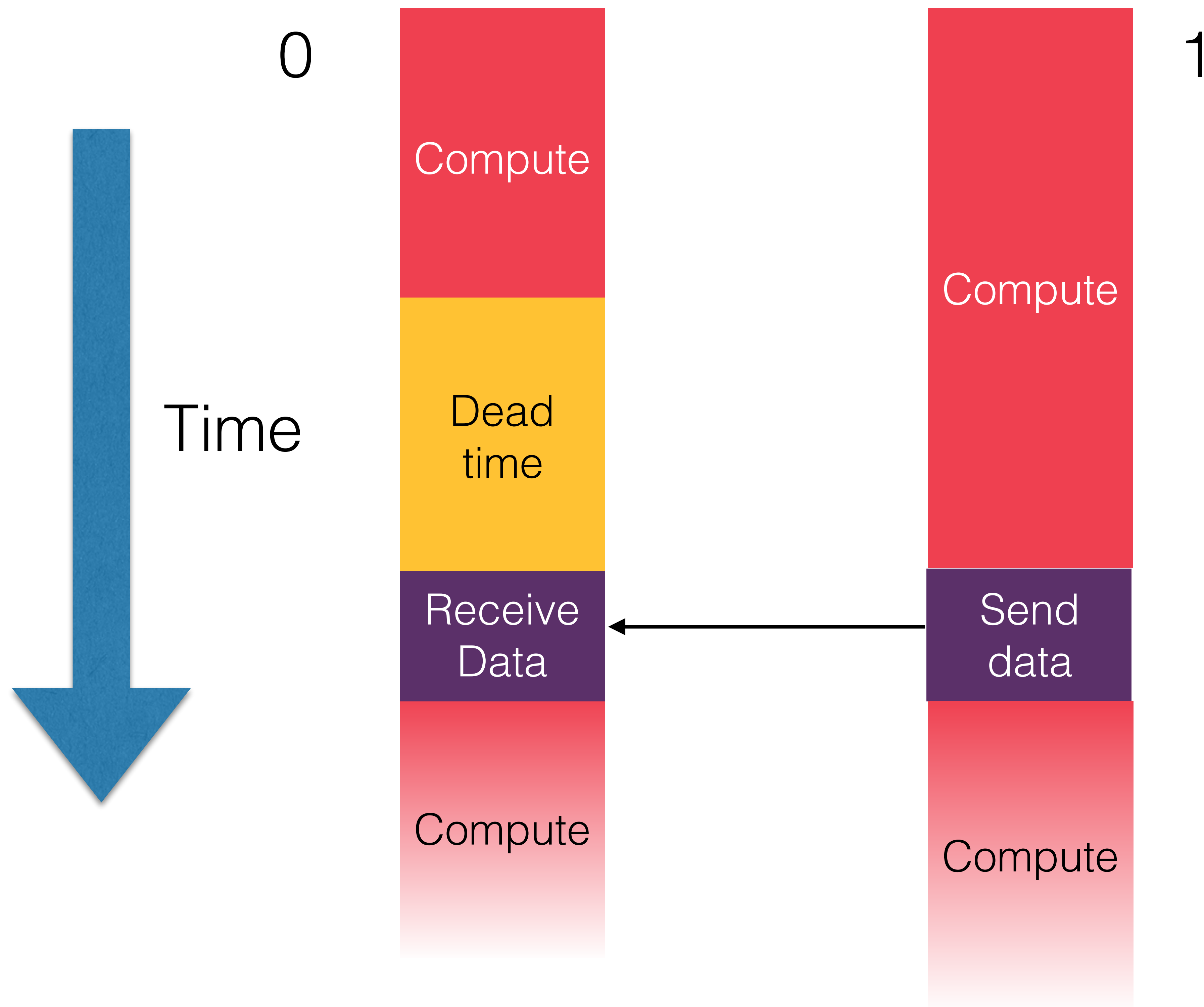
Strongly coupled



Why want explicit synchronisation?

- If some processors take longer than others to calculate results then you have a load imbalance
- If the system is still strongly coupled then there’s not much you can do in the communications
  - Have to **load balance** to try to prevent this from happening

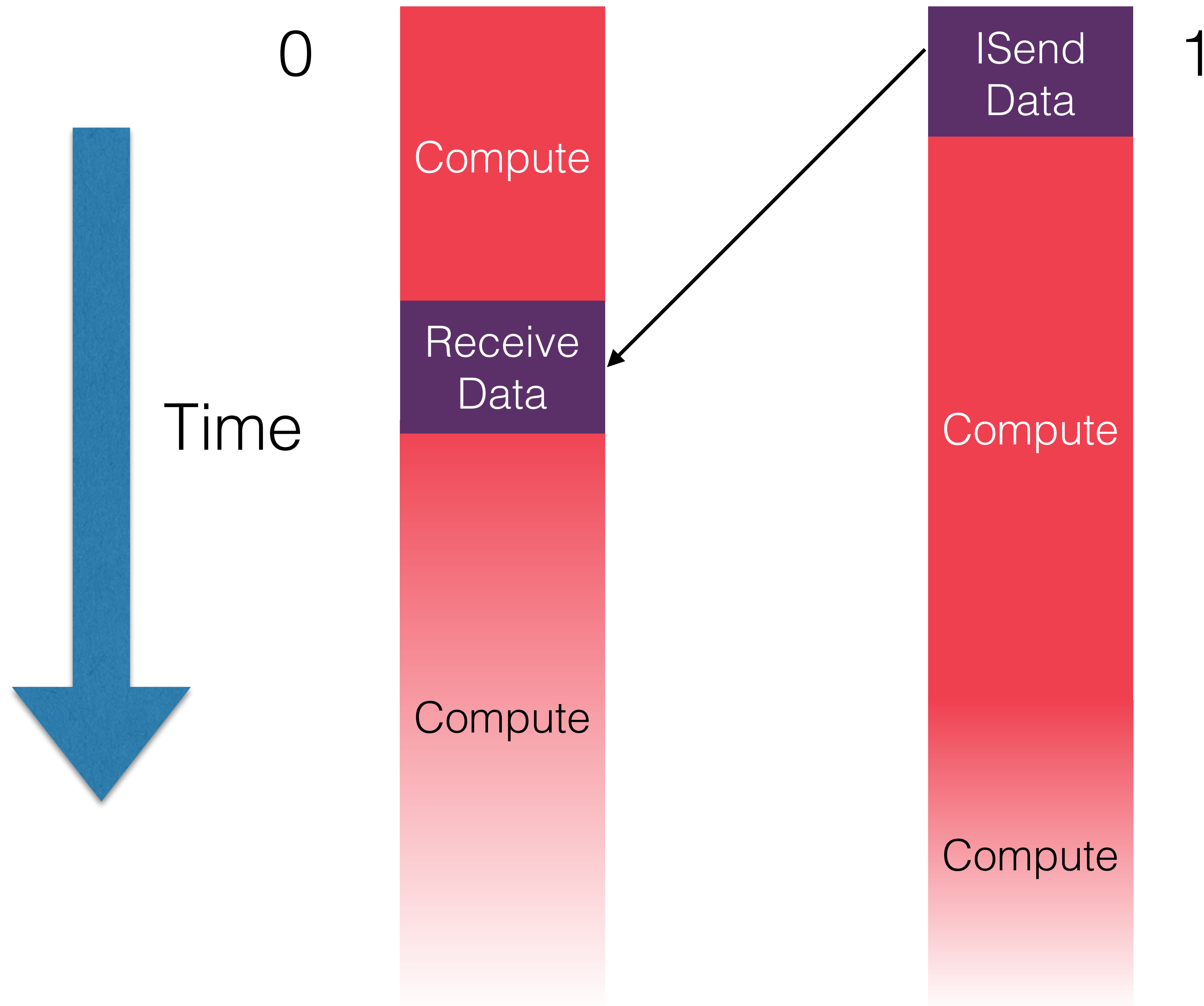
Strongly coupled but unbalanced



Why want explicit synchronisation?

- For more weakly coupled systems you might be able to access data from a remote processor as soon as you want it
- Still don't need one sided though
- Non-blocking sends and receives will do

Weakly coupled



Why want explicit synchronisation?

- If however processor 1 can’t know what data processor 0 will want before it starts its computation you can’t do this
- If processor 0 knows what it will want then you can have a negotiation phase before communication starts
  - Still locks next time you need to negotiate
- Sometimes processor 0 doesn’t know at the start either
- Better if processor 0 can just get data from processor 1 when it wants it

Undefinable requirement

0

Compute

epoch

Get Data

epoch

Compute

epoch

1

Compute

epoch

Compute

Data is available on 0 at the end of the epoch on processor 0

Correct data is available to 0 anywhere between epoch markers

Time

Performance

- Generally comparable to two sided
- **Sometimes** one sided communication can be faster than two sided communication
- True on systems with hardware support for remote memory access
  - Single nodes
  - Cray machines
  - Some support in Infiniband
    - Not usually faster than two sided

MPI one sided
Memory Windows

Concepts

- Local memory is not in general available outside the processor that created it

- To tell MPI that memory should be available to other processes, you create a “**window**”

- Sometimes distinguish between a “window” (a view on a processor’s memory) and a “window set” (the collection of all such views over all processors in a communicator)

- MPI routines themselves just talk about a window, so we’ll stick with that

- There are several ways of creating a window

MPI_Win_create

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
MPI_Win *win)
```

- base - Pointer to the start of the memory that you want to create a window into (explicit pointer in C, just the variable in Fortran). Memory must already be allocated
- size - length of memory window in bytes
- disp_unit - length of displacements in the window in bytes. Typically either "1" to treat the window as a simple byte stream, or a size derived from sizeof()
- info - MPI_Info object for hints. See https://www.open-mpi.org/doc/v2.0/man3/MPI_Win_create.3.php
- comm - The communicator that this window is to be valid on
- win - The resulting window

MPI_Win_create

- MPI_Win_create makes an existing array available for remote access
- MPI standard requires that this must work for any memory that you give it
- There **might** be "correct" ways to allocate memory on a given machine
    - Special RMA memory areas
    - Memory alignment requirements for performance

MPI_Alloc_mem

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

- `size` - Size of memory area to be allocated in bytes
- `info` - MPI_Info object. Usually MPI_INFO_NULL unless specified for a given machine
- `baseptr` - Pointer to allocated memory. Simple pointer in C, TYPE(C_PTR) or Cray pointer in Fortran
    - Must convert from C pointer to Fortran pointer using `C_F_POINTER` for other functions

MPI_Free_mem

```
int MPI_Free_mem(void *base)
```

- base - Pointer to memory allocated with MPI_Alloc_mem. Should be simple pointer in C, Fortran pointer in Fortran

MPI_Win_allocate

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,  MPI_Comm comm,
void *baseptr, MPI_Win *win)
```

- `size` - length of memory window in bytes
- `disp_unit` - length of displacements in the window in bytes. Typically either "1" to treat the window as a simple byte stream, or a size derived from typeof()
- `info` - MPI_Info object for hints.
- `comm` - The communicator that this window is to be valid on
- `base` - Pointer to a memory location to hold the pointer to the data. Should be type (thing)** in C or TYPE(C_PTR) in Fortran
- `win` - The resulting window

MPI_Win_allocate

- Acts as combined MPI_Alloc_mem and MPI_Window_create
- Do not need to free memory with MPI_Free_mem
- Memory freed when window freed

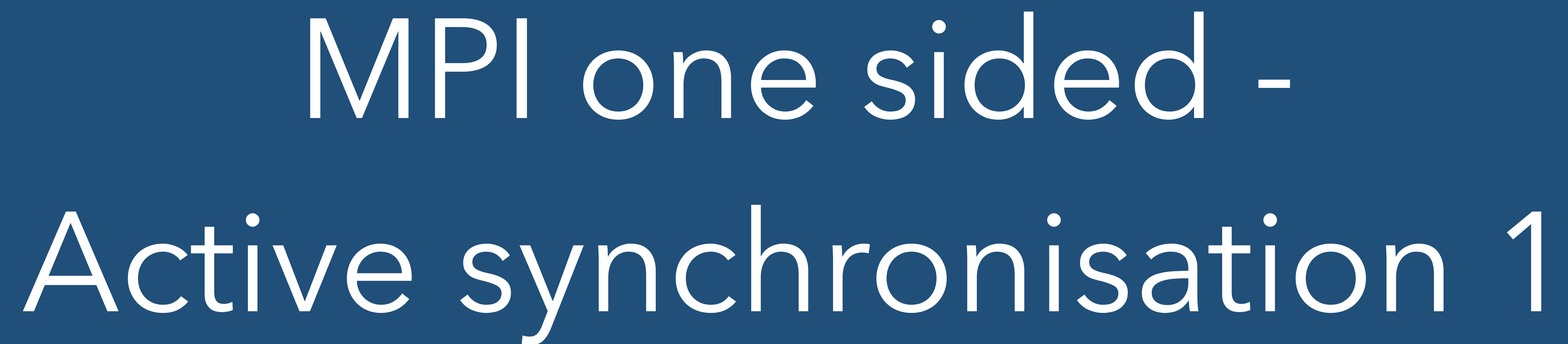
Dynamic Windows

- If you have several chunks of memory that should be in a single window, or you want memory to be freed and reallocated then you can use a dynamic window
- Newest MPI3 standard only
- `MPI_Win_create_dynamic` - creates a window
- `MPI_Win_attach` - attach memory to a window
  - Memory regions that overlap cannot be attached to the same window
- `MPI_Win_detach` - detach memory from a window
- Bit specific for this course

MPI_Win_free

```
int MPI_Win_free(MPI_Win *win)
```

- win - Window to be freed
- Frees the window and makes it invalid for further use
- Detaches any memory windows attached to the window


MPI one sided -
Active synchronisation 1

Concepts

- Once you have defined a window you have to control access to the memory
- Make sure that reads and writes only happen when they're supposed to
- MPI provides a model rather like Bulk Synchronous Parallelism (https://en.wikipedia.org/wiki/Bulk_synchronous_parallel)
- Two "epochs" that individual ranks can be separately in
  - Access epoch - can access data on other ranks
  - Exposure epoch - allows other ranks to access it's memory

Concepts

- Once the exposure epoch has started it isn't safe to write to the memory window using pointers
  - Explained more later
- Only sure that data is finally in place when the access epoch is over
  - Also explained more later

Fenceposting

- “Fenceposts” MPI access epochs
  - First call enters both “access” and “exposure” epochs
  - Second call exits both “access” and “exposure” epochs
  - Third call enters both …. etc.
  - Few caveats, but broadly true

MPI_Win_fence

```
int MPI_Win_fence(int assert, MPI_Win win)
```

- `assert` - Special conditions to optimise communication. 0 is always acceptable.
  - MPI_MODE_NOSTORE - Local memory not updated since last call to MPI_Win_fence
  - MPI_MODE_NOPUT - Local memory will not be updated by RMA **put** or **accumulate** calls. Can still use **get**.
  - https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node130.htm for others
- `win` - Window to be fenceposted

MPI one sided - Remote actions 1

MPI_Put

```
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int
target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win
win)
```

- **`origin_addr`** - Buffer for data to put on the remote rank
- **`origin_count`** - Number of **`origin_datatype`**s to put on the remote rank
- **`origin_datatype`** - Type of local data. Can be a derived type
- **`target_rank`** - Rank of destination for put. Must be the rank in the communicator specified when **`win`** was created
- **`target_disp`** - displacement from the start of the target window in units of the disp_unit specified when **`win`** was created
- **`target_count`** - number of **`target_datatype`**s to put into the window
- **`target_datatype`** - Type of data in the remote window. Can be a derived type.
- **`win`** - The window on which to perform the put. Data is put into the memory associated with the window

MPI_Get

```
int MPI_Get(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int
target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win
win)
```

- `origin_addr` - Buffer into which to receive the data from the remote rank
- `origin_count` - Number of `origin_datatype`s to get from the remote rank
- `origin_datatype` - Type of local data. Can be a derived type
- `target_rank` - Rank of source for get. Must be the rank in the communicator specified when `win` was created
- `target_disp` - displacement from the start of the target window in units of the `disp_unit` specified when `win` was created
- `target_count` - number of `target_datatype`s to get from the window
- `target_datatype` - Type of data in the remote window. Can be a derived type.
- `win` - The window on which to perform the get. Data is taken from the memory associated with the window

Example

Example

```
!MPI helper routine to get size of int
CALL MPI_Sizeof(recv_rank, intsize, ierr)
!Just using a single int here
size_of_window = intsize * n_elements

CALL MPI_Win_allocate(size_of_window, intsize, MPI_INFO_NULL, MPI_COMM_WORLD,&
    c_pointer, window, ierr)

!Get Fortran pointer to
CALL C_F_POINTER(c_pointer, f_pointer, shape=(/n_elements/))

!Populate source data object
DO iindex = 1, n_elements
  data(iindex) = iindex + rank
END DO

!Use collective synchronization model. After this command any processor
!can use MPI_Put or MPI_Get on any other processor
CALL MPI_Win_fence(0, window, ierr)

!Put the result into the first (zeroth) slot
offset = 0
!Actual call to put the data in the remote processor
CALL MPI_Put(data, n_elements, MPI_INTEGER, right, offset, n_elements, &
    MPI_INTEGER, window, ierr)

!Call Win_fence again to end the access and exposure epochs
CALL MPI_Win_fence(0, window, ierr)
!Print output
PRINT ("(a,i3, a, i3, a, 10i3)"),"Rank ", rank, " got message from rank ", &
    left, " of ", f_pointer
```

Example

```
!MPI helper routine to get size of int
CALL MPI_Sizeof(recv_rank, intsize, ierr)
!Just using a single int here
size_of_window = intsize * n_elements

CALL MPI_Win_allocate(size_of_window, intsize, MPI_INFO_NULL, MPI_COMM_WORLD,&
    c_pointer, window, ierr)

!Get Fortran pointer to
CALL C_F_POINTER(c_pointer, f_pointer, shape=(/n_elements/))

!Populate source data object
DO iindex = 1, n_elements
  data(iindex) = iindex + rank
END DO

!Use collective synchronization model. After this command any processor
!can use MPI_Put or MPI_Get on any other processor
CALL MPI_Win_fence(0, window, ierr)

!Put the result into the first (zeroth) slot
offset = 0
!Actual call to put the data in the remote processor
CALL MPI_Put(data, n_elements, MPI_INTEGER, right, offset, n_elements, &
    MPI_INTEGER, window, ierr)

!Call Win_fence again to end the access and exposure epochs
CALL MPI_Win_fence(0, window, ierr)
!Print output
PRINT ("(a,i3, a, i3, a, 10i3)"),"Rank ", rank, " got message from rank ", &
    left, " of ", f_pointer
```

Results

```
Rank   0 got message from rank  15 of  16 17 18 19 20 21 22 23 24 25
Rank   1 got message from rank   0 of   1  2  3  4  5  6  7  8  9 10
Rank   2 got message from rank   1 of   2  3  4  5  6  7  8  9 10 11
Rank   3 got message from rank   2 of   3  4  5  6  7  8  9 10 11 12
Rank   4 got message from rank   3 of   4  5  6  7  8  9 10 11 12 13
Rank   5 got message from rank   4 of   5  6  7  8  9 10 11 12 13 14
Rank   6 got message from rank   5 of   6  7  8  9 10 11 12 13 14 15
Rank   7 got message from rank   6 of   7  8  9 10 11 12 13 14 15 16
Rank   8 got message from rank   7 of   8  9 10 11 12 13 14 15 16 17
Rank   9 got message from rank   8 of   9 10 11 12 13 14 15 16 17 18
Rank  10 got message from rank   9 of  10 11 12 13 14 15 16 17 18 19
Rank  11 got message from rank  10 of  11 12 13 14 15 16 17 18 19 20
Rank  12 got message from rank  11 of  12 13 14 15 16 17 18 19 20 21
Rank  13 got message from rank  12 of  13 14 15 16 17 18 19 20 21 22
Rank  14 got message from rank  13 of  14 15 16 17 18 19 20 21 22 23
Rank  15 got message from rank  14 of  15 16 17 18 19 20 21 22 23 24
```

- Works as expected

MPI one sided -
Active synchronisation 2

Manually controlling epochs

- You can specify manual entry into and exit from each epoch.
- Called PSCW (Post/Start/Complete/Wait) or Generalised Active Target Synchronisation
- Have to introduce the concept of a collection of ranks that is not a communicator
- MPI_Group

MPI_Comm_group

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

- **`comm`** - Communicator to make group from
- **`group`** - Outputs produced group containing all ranks in **`comm`**

MPI_Group_incl

```
int MPI_Group_incl(MPI_Group group, int n, const int
ranks[], MPI_Group *newgroup)
```

- **`group`** - Existing populated group referring to ranks
- **`n`** - number of ranks to include in new group
- **`ranks`** - array of ranks to include in new group
- **`newgroup`** - Output new group

MPI_Group_free

```
int MPI_Group_free(MPI_Group *group)
```

- **`group`** - group to be freed. Can no longer validly be used after freeing

Epoch commands

- There are four commands
  - MPI_Win_start - starts the **access** epoch
  - MPI_Win_complete - end the **access** epoch
  - MPI_Win_post - start the **exposure** epoch
  - MPI_Win_wait - end the **exposure** epoch
- MPI_Win_wait will not complete until all ranks that called MPI_Win_start call MPI_Win_complete
- Data is not guaranteed to be in final position until all ranks have called the appropriate MPI_Win_complete or MPI_Win_wait calls

Remember!

- Ranks that **get** or **put** data to or from another rank must be in the **access** epoch
- Ranks that are going to have data **get** or **put** into their memory must be in the **exposure** epoch
- Ranks that do both must be in **both** epochs

MPI_Win_start

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

- `group` - group of processors to put into the **access** epoch
- `assert` - same as for `MPI_Win_fence`. 0 is always OK
- `win` - Window to start epoch on

MPI_Win_post

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

- `group` - group of processors to put into the **exposure** epoch
- `assert` - same as for `MPI_Win_fence`. 0 is always OK
- `win` - Window to start epoch on

MPI_Win_complete

```
int MPI_Win_complete(MPI_Win win)
```

- `win` - Window to end **access** epoch on

- Non blocking operation itself

- Must be called on all processors that called `MPI_Win_start` or `MPI_Win_wait` **will** block

MPI_Win_wait

```
int MPI_Win_wait(MPI_Win win)
```

- `win` - Window to end **exposure** epoch on

- Blocking operation

- Until all processors that called `MPI_Win_start` call `MPI_Win_complete` this routine will lock

MPI one sided - Remote actions 2

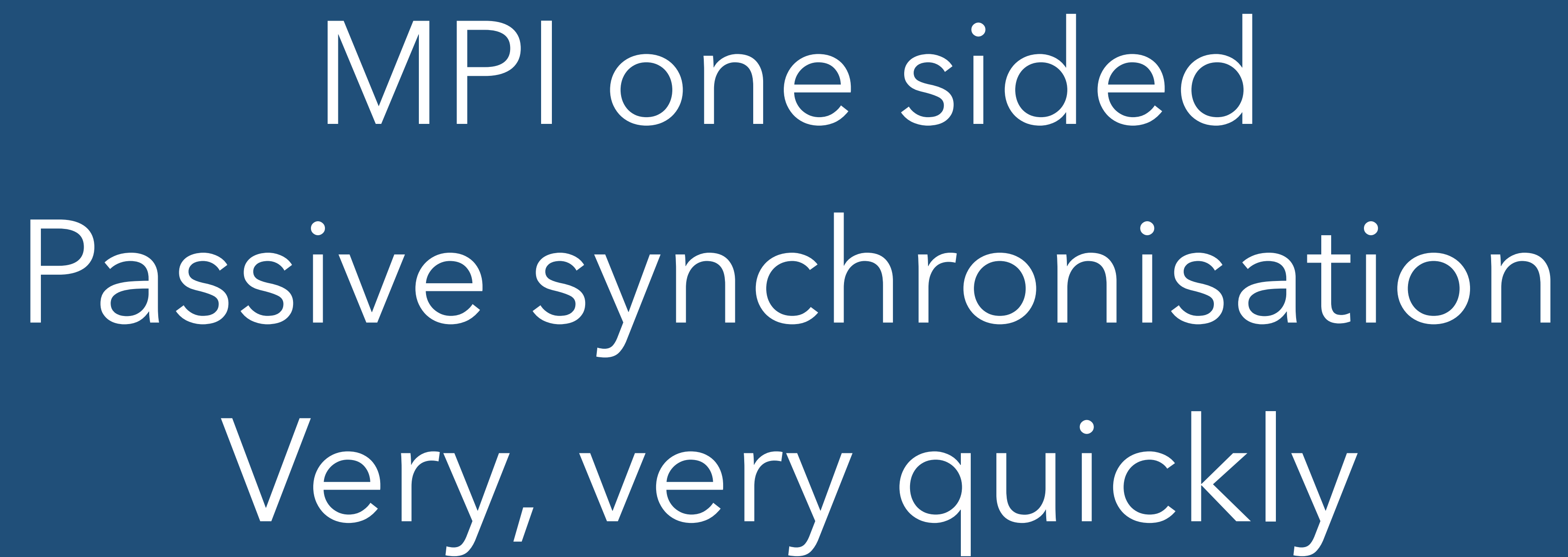
Accumulate

- Reduce for one-sided communications
- For once some guarantees about order
- Guarantees that result will be “correct”
  - All processors that call `MPI_Accumulate` with `MPI_SUM` (say) operation will have their values added to the value on `target`
  - No worry about order

MPI_Accumulate

```
int MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int
target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

- `origin_addr` – Address of source data for accumulate
- `origin_count` - Elements of origin_datatype in source
- `origin_datatype` - Type representing source. Can be custom
- `target_rank` - Rank to accumulate result into
- `target_disp` - displacement from the start of the target window in units of the `disp_unit` specified when `win` was created
- `target_count` - Elements of target_datatype in destination
- `target_datatype` - Datatype of destination on target. Can be custom
- `op` - Operation. Same as MPI_Reduce operations + **MPI_REPLACE** (replace value on target with value in origin_addr. Atomic replace operation). Cannot use user defined operations
- `win` - Window to operate on


MPI one sided
Passive synchronisation
Very, very quickly

Passive synchronisation

- Allows one rank to both put itself in the **access** epoch and put the remote processor in the **exposure** epoch
- This gives you a lot less certainty about what's going to happen
- Tricky to use, but potentially very powerful

MPI_Win_lock (MPI3)

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
```

- `lock_type`
  - `MPI_LOCK_SHARED` - other processors can also lock this window on rank
  - `MPI_LOCK_EXCLUSIVE` - other processors will be excluded from locking this window on rank
- `rank` - remote processor to put in **exposure** epoch
- `assert` - As MPI_Win_fence. 0 always acceptable
- `win` - Window to operate on

MPI_Win_unlock (MPI3)

```
int MPI_Win_unlock(int rank, MPI_Win win)
```

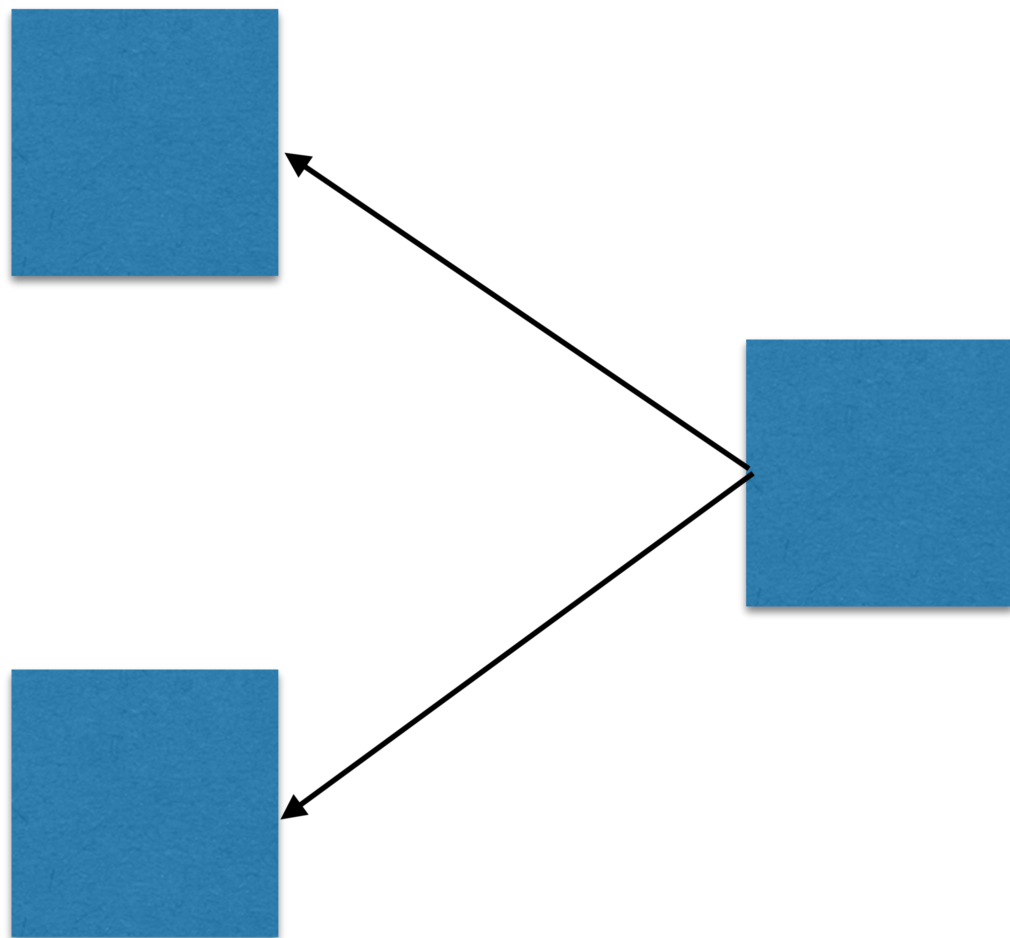
- `rank` - remote processor to remove from **exposure** epoch
- `win` - Window to operate on

Risks and complications

Risks

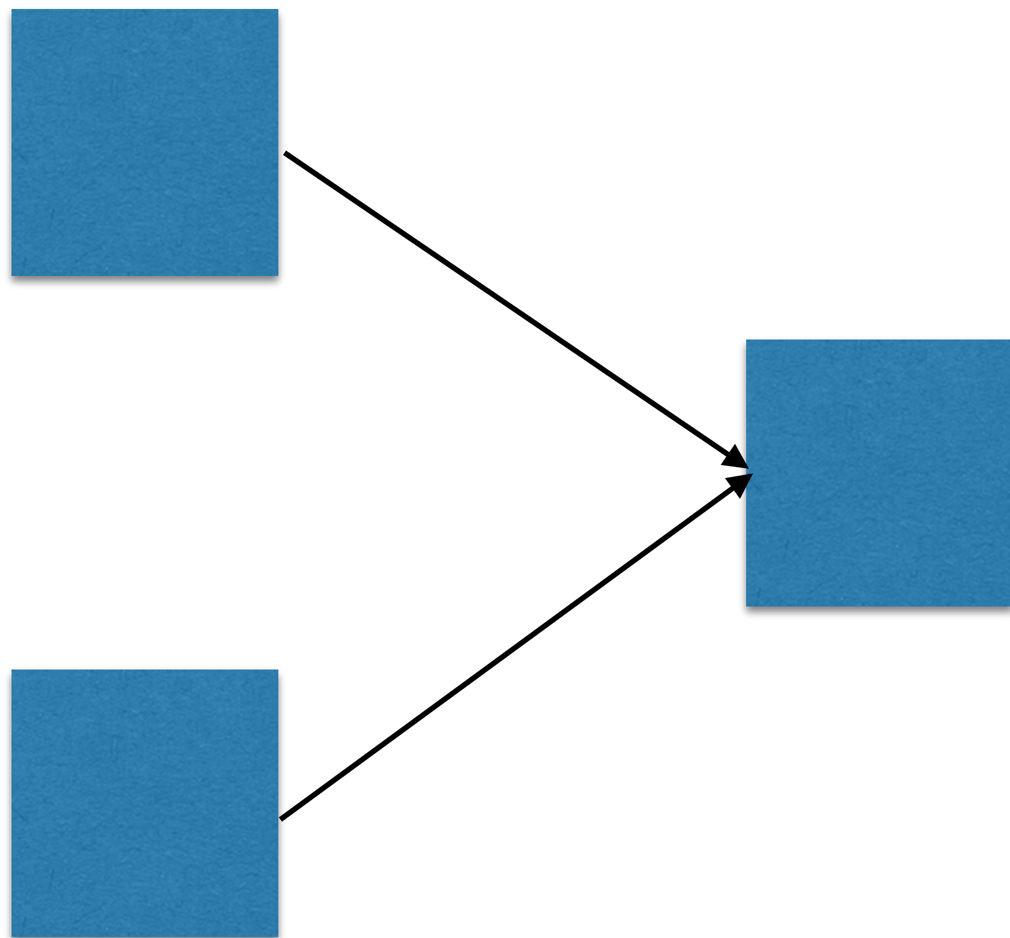
- MPI one sided communications gives fewer guarantees than regular MPI communications
- You guarantee a point at which all operations are finished
- **You have no idea of when they happen before that point**
- **Or the order in which they happen**

Risks



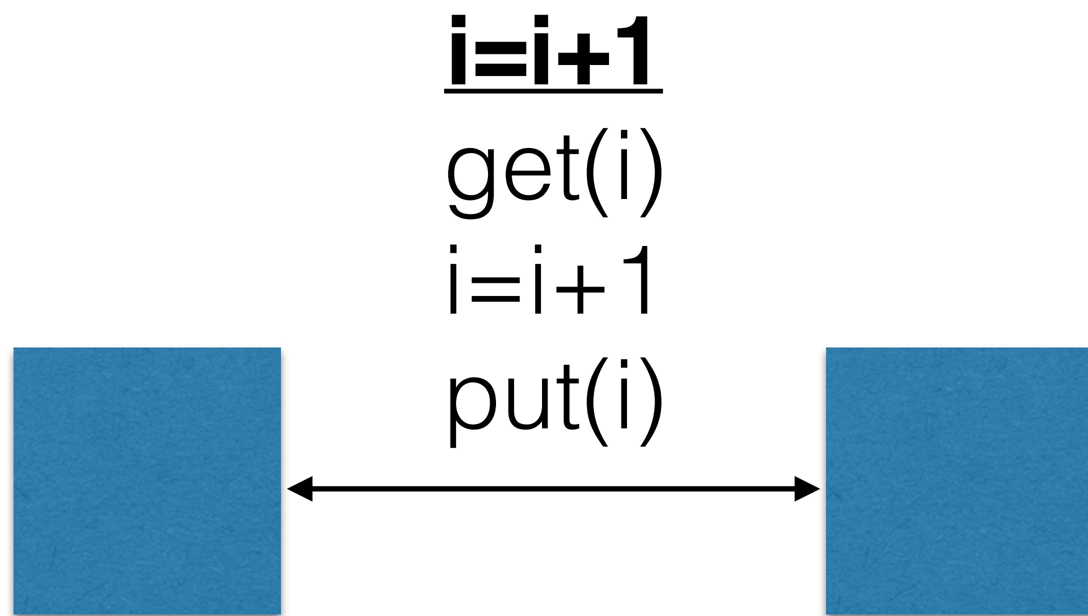
- Two ranks getting data from a remote rank works OK
- Both get the same value
- So long as the data source doesn't change it between the two calls

Risks



- Two ranks putting data onto a remote process at the same time not OK
- Undefined behaviour
- Could be either value
- Could be useless intermediate state

Risks



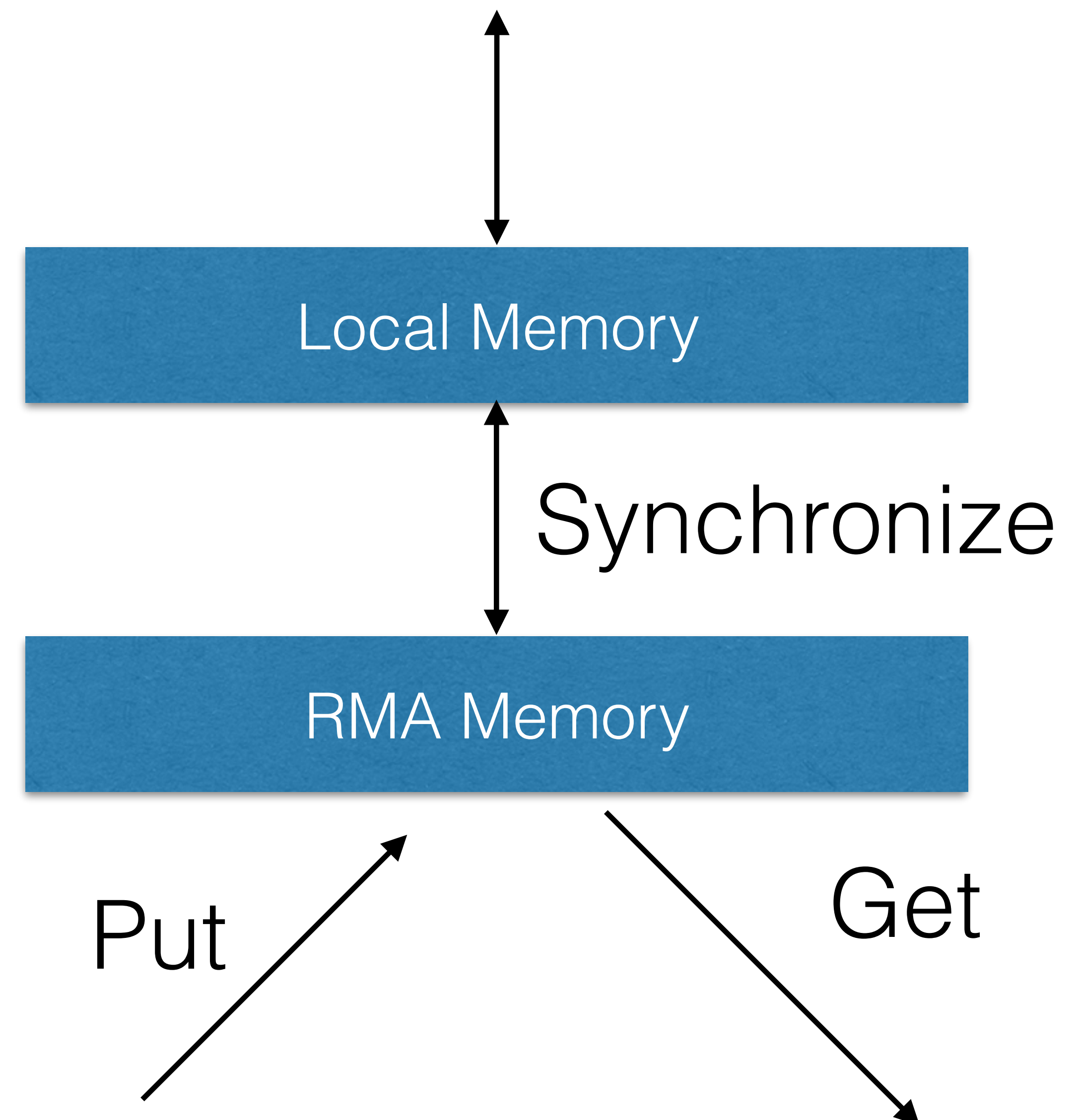
- Can't do this
- Might work as expected with get followed by put
- Might not

Risks

- So what's the complete list of what you can do and what you can't do?
- Sadly, it depends on what "**memory model**" your MPI implementation is using
- You can check using "**MPI_WIN_MODEL**" as a parameter to "**MPI_Win_get_attr**"
- Generally easier to just assume that you're using the older MPI_WIN_SEPARATE model
- Following table shows what commands can be used within a single synchronisation period

MPI_WIN_SEPARATE

- Oldest and most conservative model
- There is specific memory set aside for remote access
- It synchronises with the actual memory storing a variable at synchronisation points
  - **MPI_Win_sync(window)**
- At other times, they are not guaranteed to be the same



Safety

	Load	Store	Get	Put	Accumulate
Load					
Store				X	X
Get					
Put		X			
Accumulate		X			

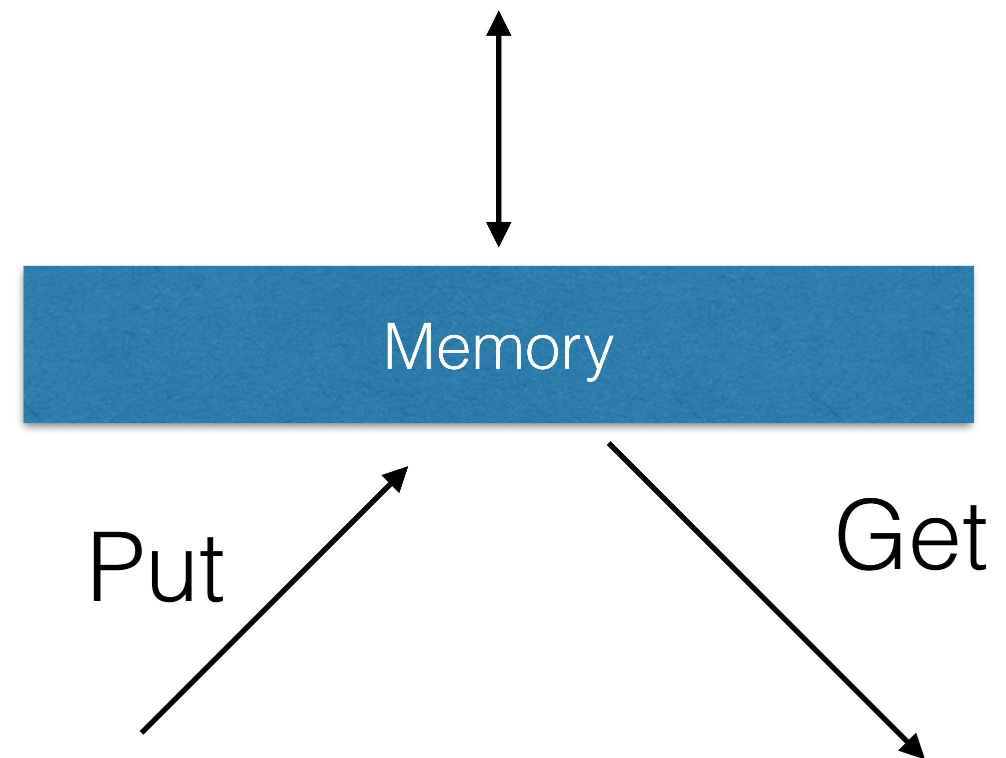
Can use on same data in window

Can use on different data in window

Cannot safely use together

MPI_WIN_UNIFIED

- Only defined in MPI3 standard
- Now is just "memory"
- All accesses, whether from local or remote sources affect the memory directly and immediately



Safety

	Load	Store	Get	Put	Accumulate
Load					
Store					
Get					
Put					
Accumulate					

Can use on same data in window

Can use on different data in window