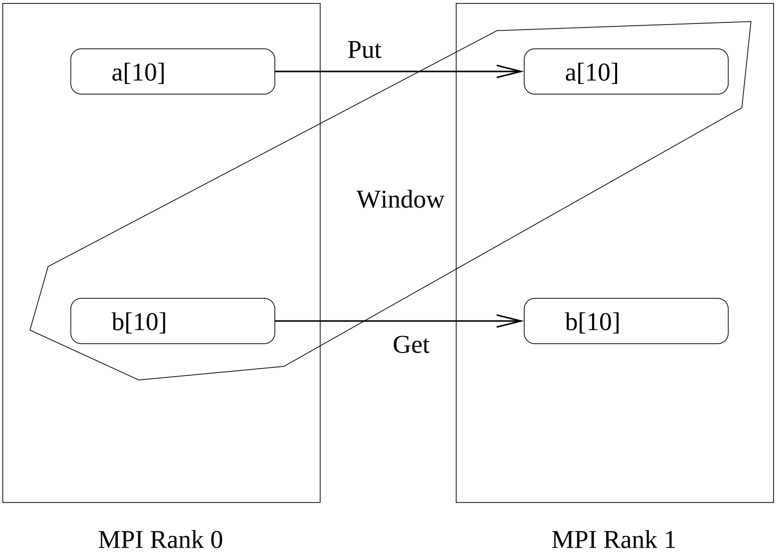
Communications Review

- Up to now we have used three sorts of communications
  - Blocking point-to-point `MPI_Send()`
  - Blocking collective `MPI_Reduce()`
  - Non-blocking point-to-point `MPI_Isend()`
- All of these are co-operative operations
- All parties in the communication need to explicity call various functions to complete the transaction

One Sided Communications

- There are times when we need access to data on a different MPI task but at a time that is not necessarily known by the other task
- Could use `MPI_Isend` but then there are problems with updates and which version of data is sent
- MPI-2 introduces the concept of one sided communications, often called remote memory access (RMA)
- We define a region of our memory (a window) that we want other processes to be able to read and then let them at it
- Other processes can use Get or Put operations to access our memory

One Sided Communications



Creating Windows

- MPI_Win_create(base, size, disp, info, comm, &win)
- This is a collective operation across `comm`
- Exposes the memory from `base` for `size` bytes to processes in `comm`
- The base is often set to `MPI_BOTTOM` on nodes where you don't have data you want to expose
- Need to be careful not to use the window on those nodes
- `disp` is used to scale the displacements between elements
  - `1` (no scaling) if there is no structure to the memory
  - `sizeof(type)` if the window is an array of `type`

Creating Windows

- `info` is an `MPI_Info` object that can be used to improve performance
- Quite often left as `MPI_INFO_NULL`
- `win` is an `MPI_Win` object used by other one sided comms in accessing the memory
- When finished with the comms we should delete the object
- MPI_Win_free(&win)

Moving Data

- Once the memory window has been created we are free to start using our RMA operation
- The standard provides us with three functions for doing this
  - `MPI_Put` — place data in a remote window
  - `MPI_Get` — retreive data from a remote window
  - `MPI_Accumulate` — update data in a remote window
- Note - all of these functions are non-blocking

MPI_Put

- MPI_Put(ldata, lcount, ltype, dest, disp, rcount, rtype, win)
- `ldata`, `lcount` and `ltype` are exactly as in `MPI_Send`
- `dest` is the rank of the process who's memory we are updating
- The location on the remote side is specified by an offset given in `disp` as well as `rcount` and `rtype` which as used like in `MPI_Recv`

MPI_Get

- MPI_Get(ldata, lcount, ltype, dest, disp, rcount, rtype, win)
- Pretty much identical to `MPI_Put`
- Remember the l values are on the side issuing the get call

MPI_Accumulate

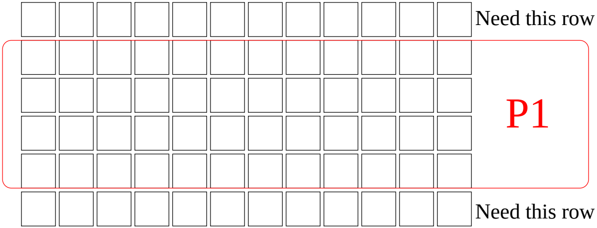
- MPI provides us with a function that merges a data move and combine
- MPI_Accumulate(ldata, lcount, ltype, dest, disp, rcount, rtype, op, win)
- Again arguments similar to Put and Get calls
- In this case the operation `op` is applied between the local data and the values already on the remote process

Completing RMA actions

- Recall we said that RMA operations are non-blocking
- Therefore we need a way to ensure that all pending operations have been completed (cf `MPI_Wait()`)
- MPI_Win_fence(assert, win)
- This is a collective operation across all tasks sharing `win`
- For the moment we will just leave `assert` as 0

Halo exchange

- Recall solving the heat equation
- We have a halo exchange between neighbouring processes
- Used one of
  - Cascade of Send/Recv
  - Checkerboard (black/white) update
  - MPI_Sendrecv
  - Non blocking Send/Recv



Halo exchange

- Instead we can replace all this complication with RMA operations
- Easiest method is to expose all of each process' grid in a window
- Alternatively create two windows — one for the rank above and one for the rank below
- Again use `MPI_BOTTOM` for the windows on the two edge processes unless you have periodic boundary conditions

Multiple Windows

- Again in the heat equation, each node actually had two grids and swapped back and forth between them
- Even if we swap pointers, once the windows are created the point to the original memory locations
- Therefore have to create a window for each section of memory, one on each grid
- We can then update another pointer to type `MPI_Win` at each iteration as we change the active grid

Asserts

- `MPI_Win_fence` takes two arguments, `assert` and `win`
- Up until now we have just set assert to 0
- `assert` can actually be a combination of up to four macros
  - `MPI_MODE_NOSTORE` — no local updates since last fence
  - `MPI_MODE_NOPUT` — no remote updates until next fence
  - `MPI_MODE_NOPRECEDE` — no RMA since last fence
  - `MPI_MODE_NOSUCCEED` — no RMA until fence
- It is good practice to use these asserts where possible as it may improve performance

RMA Locks

- Using `MPI_Win_fence` to complete RMA operations is often called active target synchronization
- Often we do not want the remote process to have to make any calls to the library once the window has been created
- This is called passive target synchronization and is implemented using `MPI_Win_lock` and `MPI_Win_unlock`
- This provides truly one sided communications

RMA Locks

- MPI_Win_Lock(lock_type, rank, assert, win)
- `lock_type` can be either `MPI_LOCK_SHARED` or `MPI_LOCK_EXCLUSIVE`
- `SHARED` is usually used when doing gets
- `EXCLUSIVE` is usually required when doing puts
- The lock only acts on the memory of the specified `rank` within the window `win`
- You need to lock your own window when doing local updates!

Scalable Synchronization

- `MPI_Win_fence` is a collective operation
- When the number of nodes grows this becomes very expensive to execute
- While locks are not collective, every request has to query the whole window to ensure no other task is trying to lock the same region
- Again, with a large number of tasks, this operation becomes very expensive
- MPI-2 has another set of synchronization functions for RMA that are only called on participants in the RMA operation

Scalable Synchronization

- We define an *exposure epoch* to be the time that a process exposes its window to RMA operations from other tasks
- This epoch starts with a call to `MPI_Win_post` and ends with a call to `MPI_Win_wait`
- An *access epoch* is the time when a task wants to use RMA operations to access another task's window
- This epoch starts with a call to `MPI_Win_start` and ends with a call to `MPI_Win_complete`

Exposure Epoch

- MPI_Win_post(from_grp, assert, win)
- We use calls such as `MPI_Group_incl` to create an MPI_Group that contains the tasks that are going to use the window during the epoch
- The assert can have 3 values but by and large they are not used
  - `MPI_MODE_NOSTORE`
  - `MPI_MODE_NOPUT`
  - `MPI_MODE_NOCHECK`
- MPI_Win_wait(win)
- Blocks until a the other processes in the group in the post call have indicated they are finished

Access Epoch

- MPI_Win_start(to_grp, assert, win)
- Similarly we create a group of tasks that will be the target of RMA calls from this task
- Assert can only take `MPI_MODE_NOCHECK` but it rarely used and requires the matching `post` to have also asserted `MPI_MODE_NOCHECK`
- MPI_Win_complete(win)
- Informs the other processes in the group that we have finished carrying out RMA operations