The OpenSHMEM PGAS Communications Library

ARCHER Technical Forum, Wed 28th May 2014

David Henty <d.henty@epcc.ed.ac.uk>

Welcome!

Technical Forum talk starts at 15:00 BST

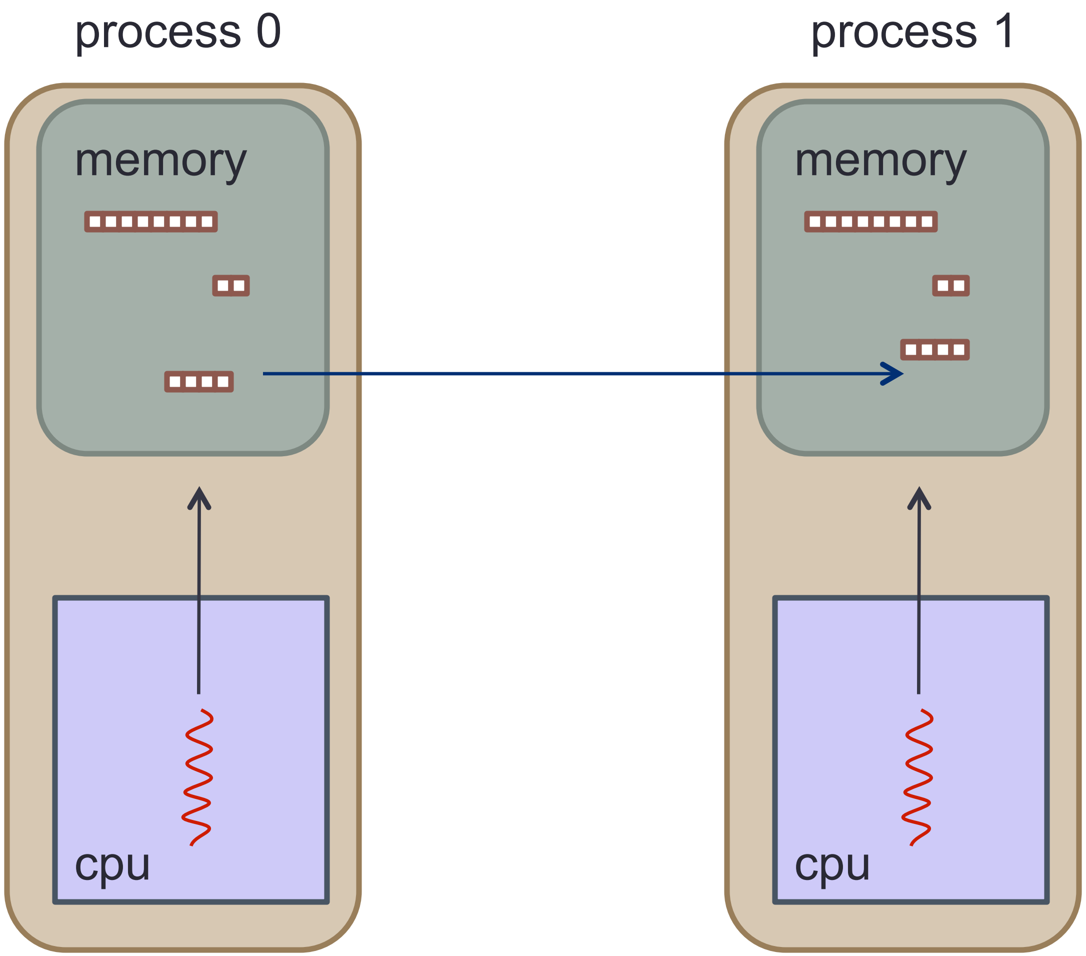
archer

epcc

THE UNIVERSITY OF EDINBURGH

Reusing this material

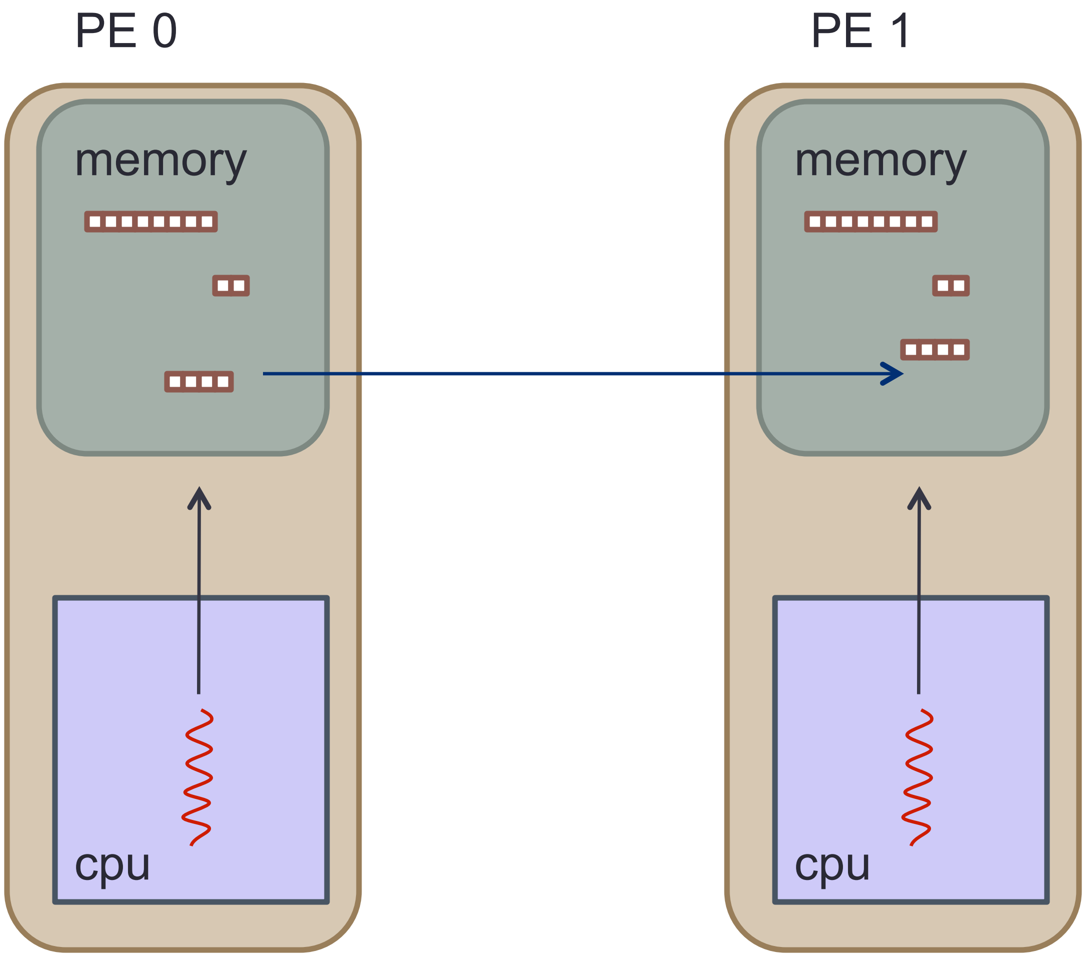
MPI



MPI_Send(a,...,1,…)

MPI_Recv(b,...,0,…)

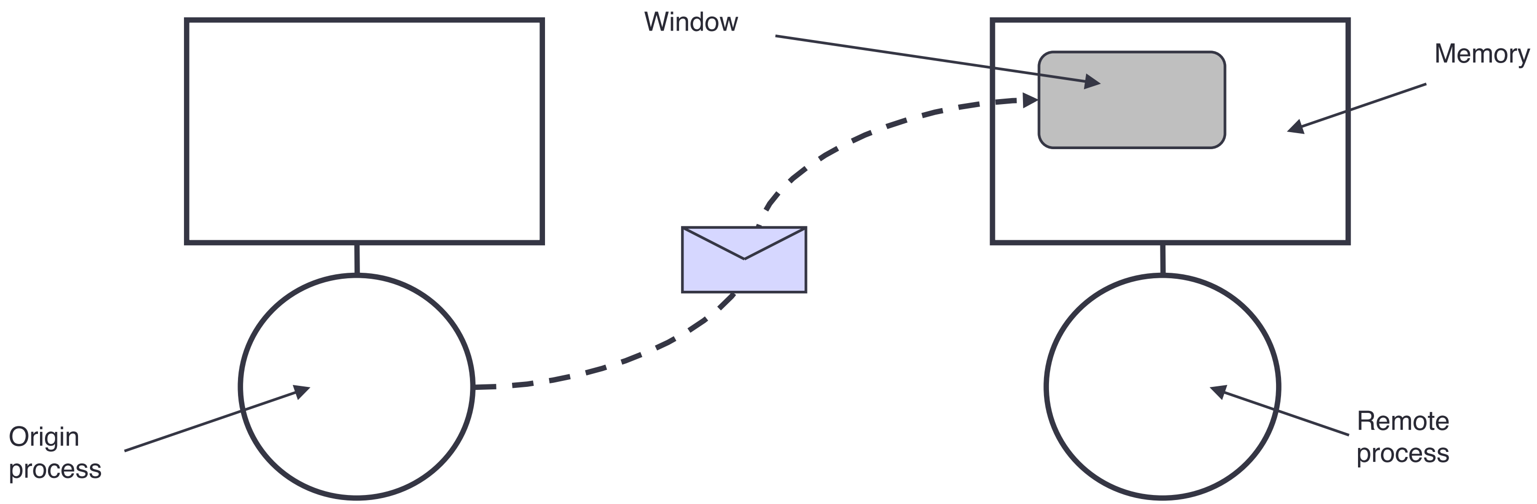
SHMEM



`shmem_put(a, b, 1, …)`

Single-Sided Model

- Remote memory can be read or written directly using library calls



- Remote process does not actively participate
  - No matching receive (or send) needs to be performed
  - Synchronisation is now a major issue
  - May be difficult to calculate remote addresses

Motivation

- Why extend the basic message-passing model?
- Hardware
  - Many MPPs support *R*emote *M*emory *A*ccess (RMA) in hardware
  - This is the fundamental model for SMP systems
  - Many users have started to use RMA calls for efficiency
    - Has lead to the development of non-portable parallel applications
- Software
  - Many algorithms naturally single-sided
    - e.g., sparse matrix-vector
  - Matching send/receive pairs requires extra programming
  - Even worse if communication structure changes
    - e.g., adaptive decomposition

History (official)

- Cray SHMEM (MP-SHMEM, LC-SHMEM)
  - Cray first introduced SHMEM in 1993 for its Cray T3D systems.
  - Cray SHMEM was also used in other models: T3E, PVP and XT
- SGI SHMEM (SGI-SHMEM)
  - Cray Research merged with Silicon Graphics (SGI) February 1996.
  - SHMEM incorporated into SGI's Message Passing Toolkit (MPT)
- Quadrics SHMEM (Q-SHMEM)
  - an optimised API for the Quadrics QsNet interconnect in 2001
- First OpenSHMEM standard in 2012

History (unofficial)

- SHMEM library developed for Cray T3D in 1993
  - basis of Cray MPI as developed by EPCC
  - many users called the SHMEM library directly for performance
  - very hard to use correctly (e.g. manual cache coherency!)
- Continued on Cray T3E
  - easier to use as cache coherency is automatic
  - possibility of smaller latencies than (EPCC-optimised) Cray T3E MPI
- Maintained afterwards mainly for porting existing codes
  - eg from important US customers such as ORNL
  - although performance on SGI NUMA machines presumably good
- OpenSHMEM an important standardisation process
  - although rather messy in places

OpenSHMEM Terminology

- PE
  - a Processing Element (i.e. process), numbered as 0, 1, 2, …, N-1
- origin
  - Process that performs the call
- remote_pe
  - Process on which memory is accessed
- source
  - Array which the data is copied from
- target
  - Array which the data is copied to

Puts and Gets

- Key routines
- PUT is a remote write
- GET is a remote read

Puts and Gets

- Key routines

> How do we know it is safe to overwrite `target`?

- PUT is a remote write
  - generically: `put(target,source,len,remote_pe)`
  - write `len` elements from `source` on origin to `target` on `remote_pe`
  - returns *before* data has arrived at target

> How do we know `source` is ready to be accessed?

- GET is a remote read
  - generically : `get(target,source,len,remote_pe)`
  - …but data is transferred in the opposite direction
  - read `len` elements from `source` on `remote_pe` to `target` on origin
  - returns *after* data has arrived at target

Remote Addresses

- In general, each process has its own local memory
- Even in SPMD, each instance of a particular variable on different processors may have a different address
  - not all processes may even declare a particular array at runtime
- It is possible for processors to access remote memory by
  - Ensuring all variable instances have the same relative address
  - Publishing variables as available for RMA
  - Publishing windows of memory as available for RMA
- OpenSHMEM takes the first approach

Symmetric Memory

- Consider `put(target,source,len,remote_pe)`
  - all parameters provided by the origin PE
  - but `target` is to be interpreted at the `remote_pe`

- Solution
  - ensure address of `target` is the same on every PE
  - not possible for data allocated on the stack or dynamically (e.g. via malloc)
  - in OpenSHMEM it must be allocated in *symmetric memory*

- Symmetric objects
  - Fortran: any data that is saved
  - C/C++: global/static data
  - or call a special version of malloc

Data Allocation

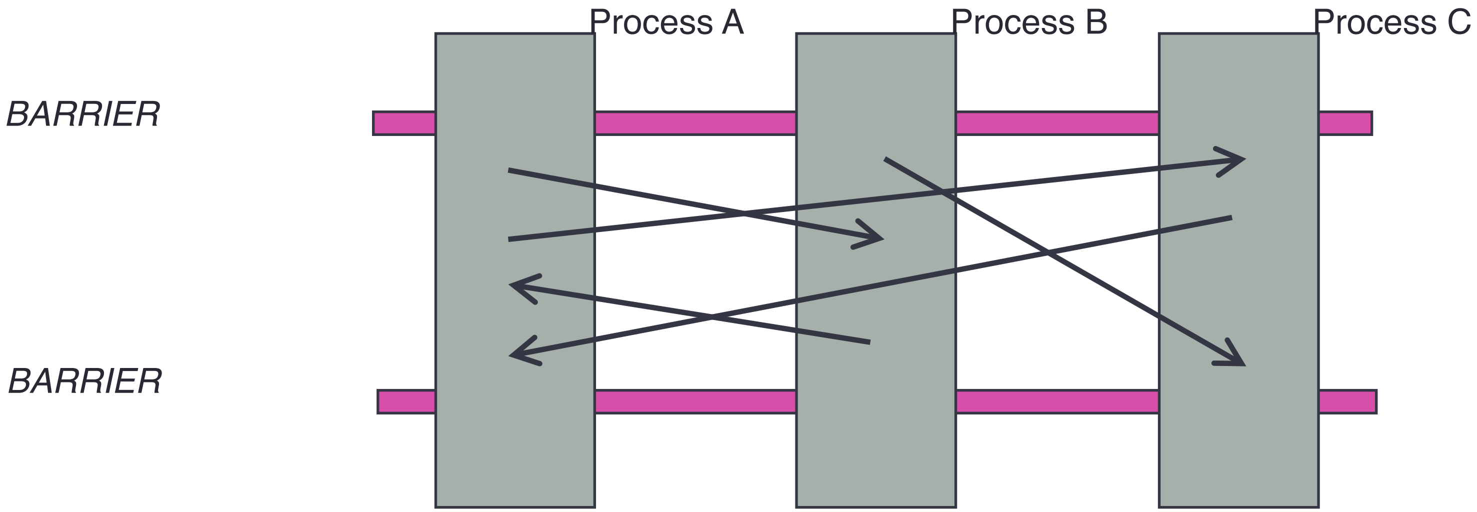
```
! Fortran
subroutine fred
  real :: x(4,4)          ! not symmetric
  real, save :: x(4,4) ! symmetric
  …
end subroutine fred

// C
float x[4][4];             // symmetric

void fred()
{
  float x[4][4];           // not symmetric
  …
}
```

Synchronisation is critical for RMA

- Various different approaches exist
  - Collective synchronisation across all processors
  - Pairwise synchronisation
  - Locks
- Flexibility needed for different algorithms/applications
  - Differing performance costs
- Synchronisation issues can become very complicated
  - Vendor-specific RMA libraries can require complex synchronisation
  - EPCC taught (correct) use of SHMEM for the T3D/T3E
    - saw many codes that worked in practice, but were technically incorrect!
- Ease-of-use sacrificed for performance

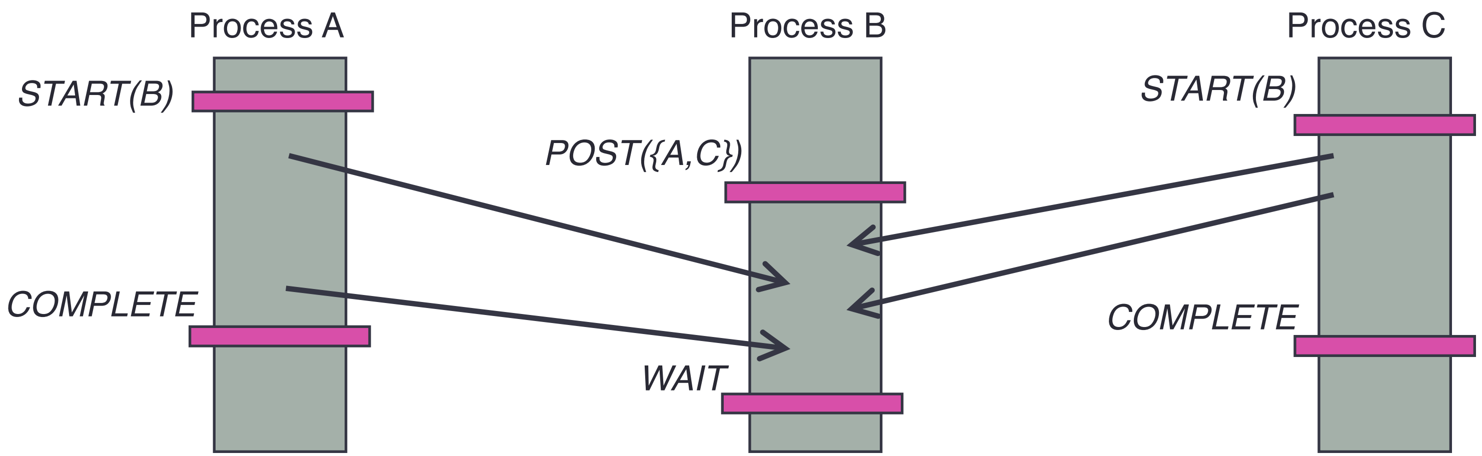
1) Collective

- Simplest form of synchronisation
- Pair of barriers encloses sequence of RMA operations
  - 2nd call only returns when all communications are complete
  - Useful when communications pattern is not known
  - Simple and robust programming model



2) Pairwise Model

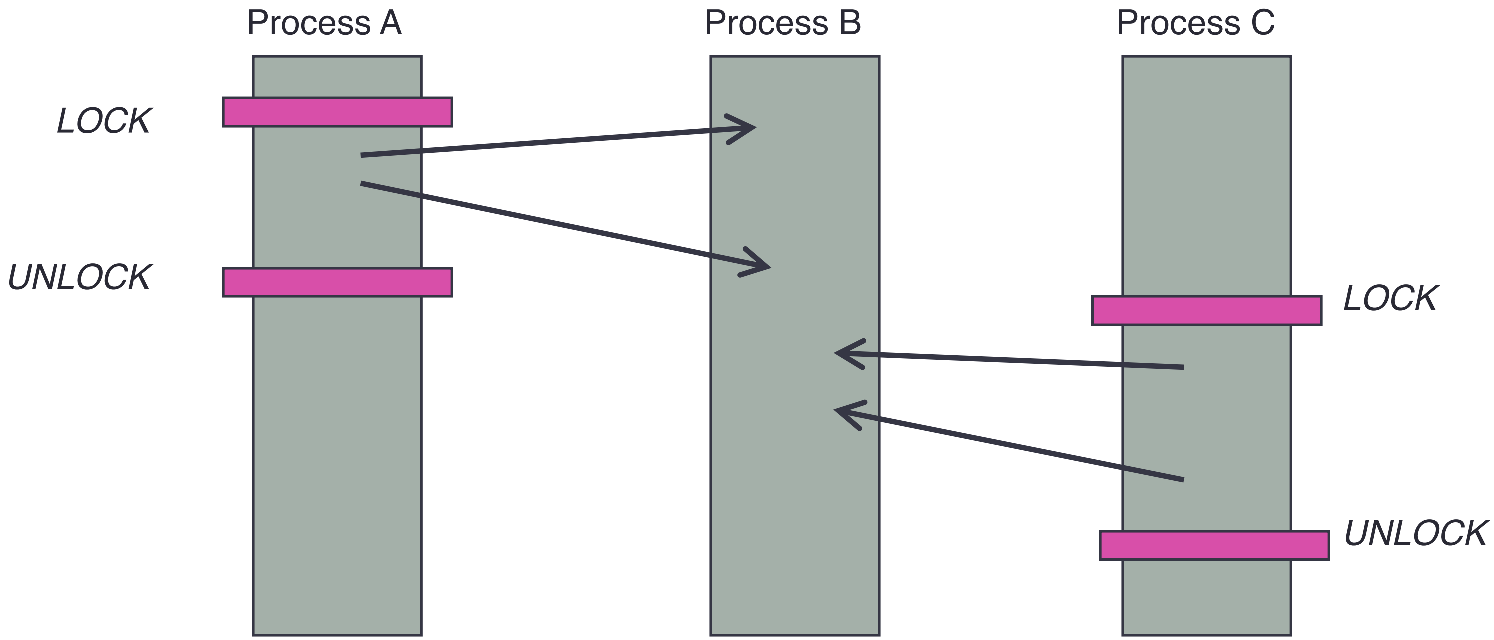
- Useful when comms pattern is known in advance
- Implemented via library routines or flag variables



- More complicated model
  - Closer to message-passing than previous collective approach
  - But can be more efficient and flexible

3) Locks

- Remote process neither synchronises nor communicates
- Origin process locks data on remote process
  - Exclusive locks ensure sequential access



OpenSHMEM PUT

- `shmem_[funcname]_put(target,source,len,remote_pe)`
  - Writes `len` elements of contiguous data from address `source` on the origin PE to address `target` on `remote_pe`
  - **`target`** must be the address of a *symmetric data object*

- Fortran
  - **`[funcname]`** can be: INTEGER, REAL, DOUBLE, COMPLEX, LOGICAL or CHARACTER}
  - e.g. `CALL SHMEM_REAL_PUT(x, y, 1, 5)`
- C
  - **`[funcname]`** can be: int, long, longlong, float, double, longdouble or char
  - e.g. `shmem_float_put(&x, &y, 1, 5)`

OpenSHMEM GET

- `CALL SHMEM_[funcname]_GET(target,source,len,remote_pe)`
  - Reads `len` elements of contiguous data from address `source` on `remote_pe` to address `target` on origin PE
  - **`[funcname]`** can be: INTEGER, DOUBLE, COMPLEX, LOGICAL, REAL or CHARACTER
  - **`source`** must be the address of a *symmetric data object*

- Similar range of routines as for PUT
  - **`SHMEM_GET32, SHMEM_INTEGER_GET`**, …

- Similar interfaces for C routines
  - e.g., **`void shmem_int_get(int *target, const int *source, size_t nelems, int remote_pe);`**

OpenSHMEM on ARCHER

- Part of the Cray Message-Passing Toolkit

```
user@archer> module load cray-shmem
```

Support Routines (Fortran)

- All Fortran programs include the header file 'shmem.fh'

- Initialisation: **`CALL START_PES(0)`**
  - Initialises the OpenSHMEM library
    - e.g., sets up the symmetric heap, PE numbers, …
  - Must be called before any other library routine is called
    - on the Cray, also need to call **`SHMEM_FINALIZE()`** at the end

- Query Routines
  - **`SHMEM_MY_PE()`** (or **`MY_PE()`**)
    - Returns the PE number of the calling PE
  - **`SHMEM_N_PES()`** (or **`NUM_PES()`**)
    - Returns the number of processing elements used to run the application

Fortran “Hello World”

```
PROGRAM Hello_World
  IMPLICIT NONE
  INCLUDE 'shmem.fh'

  INTEGER me, npes

  CALL START_PES(0)
  me   = SHMEM_MY_PE()
  npes = SHMEM_N_PES()

  WRITE(*,*) 'I am PE ', me, ' out of ', npes

  CALL SHMEM_FINALIZE()

END PROGRAM Hello_World
```

Support Routines (C)

- All C programs include the header file 'shmem.h'

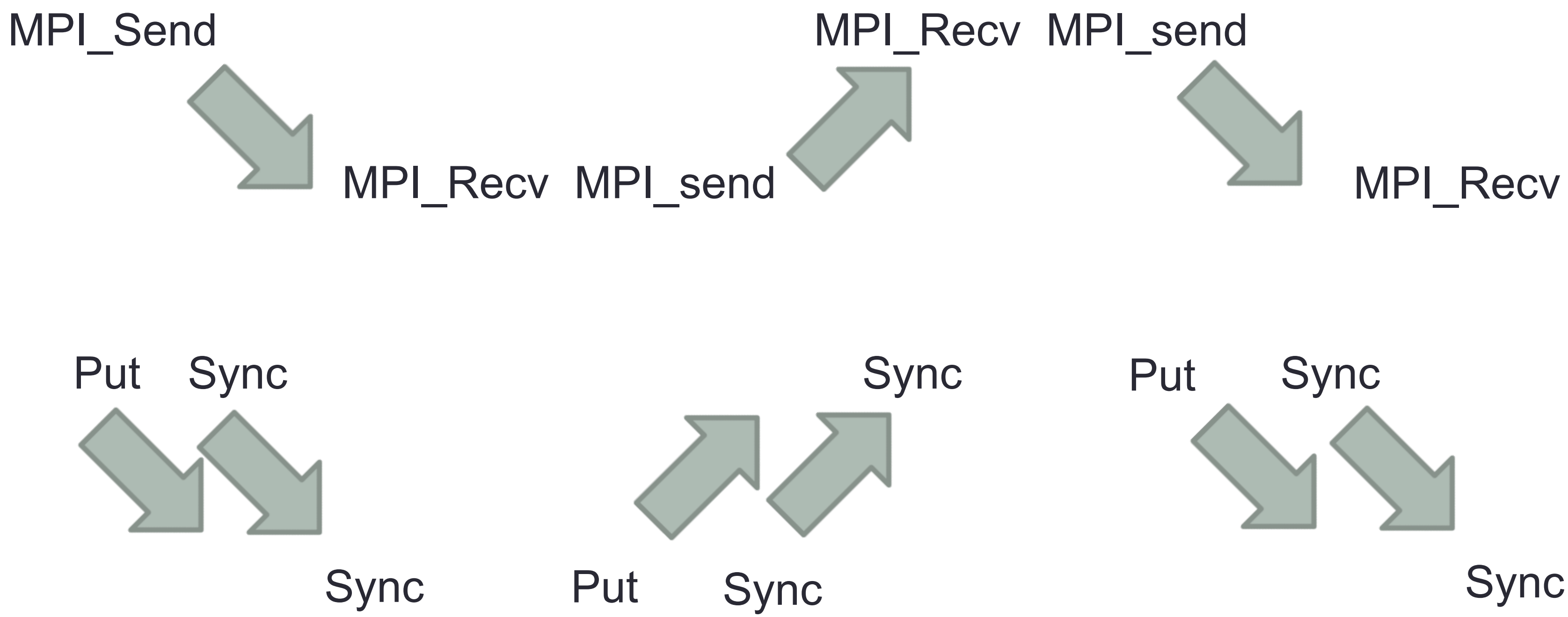
- Initialisation: `void start_pes(0);`
  - Initialises the OpenSHMEM library
    - e.g., sets up the symmetric heap, PE numbers, …
  - Must be called before any other library routine is called
    - on the Cray, also need to call `shmem_finalize()`

- Query Routines
  - `int shmem_my_pe();` (or `int _my_pe();`)
    - Returns the PE number of the calling PE
  - `int shmem_npes();` (or `int _num_pes();`)
    - Returns the number of processing elements used to run the application

C “Hello World”

```
#include "shmem.h"

int main(void)
{
 int me, npes;

 start_pes(0);

 me   = shmem_my_pe();
 npes = shmem_n_pes();

 printf("I am PE %d out of %d\n", me, npes);

 shmem_finalize();
}
```

Synchronisation for Ping-Pong



- In PGAS, synchronisation is the responsibility of the user
  - typically need **two** explicit synchronisations per communication
  - (a) is receiver ready? (b) have I received data from the sender?
  - pingpong is a special case

Global Synchronisation

```
CALL SHMEM_BARRIER_ALL()
void shmem_barrier_all();
```

- Suspend execution on the calling PE until all other PEs reach this point of execution path
  - i.e., synchronise all PEs
  - also ensures all outstanding OpenSHMEM puts are complete
- Simplest form of synchronisation
  - perhaps not the most efficient – see later

Communications details

- Vary between PGAS implementations but for OpenSHMEM:
- `put(target,source,len,remote_pe)`
  - on return, source is in the network on its way to remote pe
    - source can therefore be safely overwritten at origin pe
    - but is not guaranteed to have arrived at destination
- `get(target,source,len,remote_pe)`
  - on return, contents of source written to target on origin pe
    - target can therefore be safely read at origin pe
- So synchronisation is simpler for gets?

Using barriers

```
! wait until target is ready to receive
shmem_barrier_all
! write to remote pe
shmem_put(remote, local, ndata, target_pe)
! wait until incoming puts have completed
shmem_barrier_all

! wait until target data is ready to be read
shmem_barrier_all
! read from remote pe
shmem_get(local, remote, ndata, target_pe)
! wait until other pes have read my data
shmem_barrier_all
```

Common mistakes

- Comparison with MPI
  - If you have MPI barriers in your code that you think are required for program correctness then most probably:
    - you are either mistaken (i.e. it will run correctly and faster without barriers)!
    - or you have a bug in your code that just *happens* to disappear when you introduce barriers
  - MPI barriers are **almost never** required for correctness

- For OpenSHMEM
  - If you **do not** have synchronisation before and after puts and gets
    - you probably have an incorrect program – you will need to think **very hard** to ensure that it is correct
    - just because it *happens* to run correctly does not mean it is correct!
  - Synchronisation is **almost always** required both before and after OpenSHMEM puts and gets

Summary

- Single-sided communication invaluable for problem classes
  - Determined by the algorithm
- Simpler protocol can bring performance benefits
  - But requires thinking about synchronisation, remote addresses,...
- Various single-sided implementations now exist
  - MPI-2: quite general and portable to most platforms
  - OpenSHMEM: more limited functionality but often better performance
- Synchronisation is critical and easy to get wrong
  - As with all PGAS languages
  - Barriers are simplest OpenSHMEM approach
  - Point-to-point synchronisation also possible
    - “put data” then “put flag”