

SINGLE-SIDED PGAS COMMUNICATIONS LIBRARIES

Advanced use of OpenSHMEM



Outline

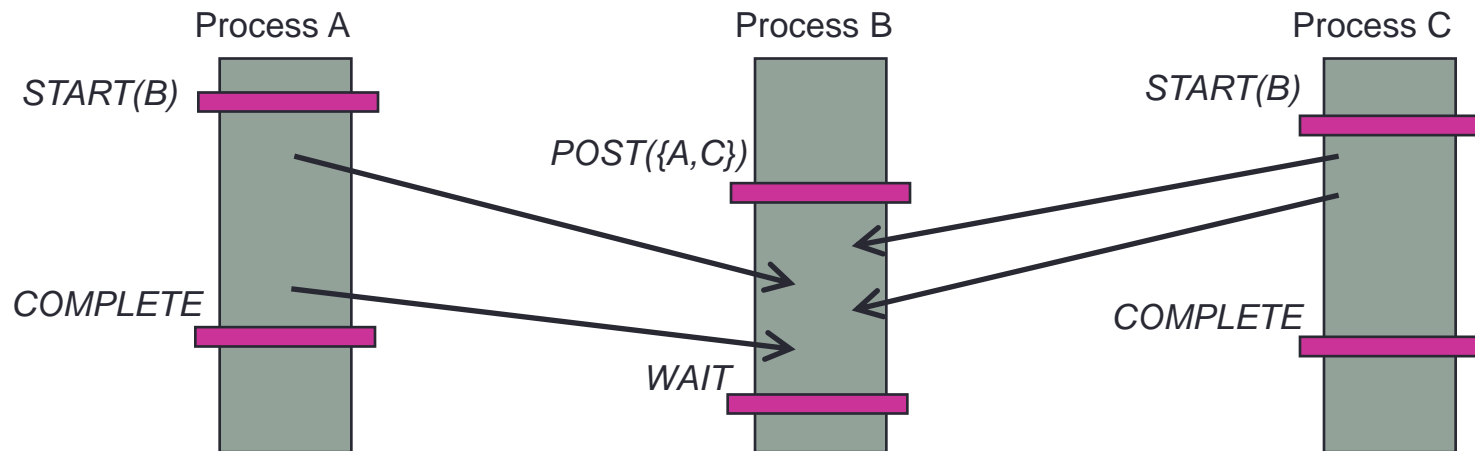
- Point-to-point synchronisation
- Collectives
- Strided transfers
- Dynamic symmetric memory allocation
- Locks and atomic updates

Point-to-point synchronisation

- Barrier synchronisation works in simple cases, but ...
- Performance issues
 - will not scale to large numbers of PEs
 - overkill in many situations
 - e.g. in traffic model, only need to synchronise with neighbours
- May not be sensible to use barriers
 - what if communications is only between a few PEs?
 - why should all PEs wait when most are not communicating?

2) Pairwise Model

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



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

OpenSHMEM idiom

- Origin PE
 - perform communication
 - write a flag variable to indicate completion
- Remote PE
 - wait until flag variable is written
 - can then access data (put) or modify buffer (get)
- Seems simple but ...
 - how do we make sure the flag arrives after the data (for put)?
 - how do we make sure that the flag is reread from memory at the remote PE and not optimised away by the compiler?

Fence and wait

- Origin PE

```
put(target, source, len, remote_pe)
shmem_fence()
put(flag, flagvalue, len, remote_pe)
```

Order of arrival not guaranteed,
e.g. dynamic routing on XC30

Ensures ordering of puts to
remote_pe before and after fence

- Remote PE (assume flag is initialised to **defaultvalue**)

```
shmem_wait(flag, defaultvalue)
```

Wait until flag differs
from **defaultvalue**

Simple spin-loop may be optimised away

Notes

- Ensuring initialisation of flag may require synchronisation
- Can also encode information in flag
 - e.g. initialise to -1
 - write the identifier of the origin PE to flag
 - `remote_pe` now knows where the data came from
- Fence works pairwise between PEs
 - can also call `shmem_quiet()`
 - waits until **all** outstanding puts from origin have **completed**
 - not usually needed
- **Not** sufficient to have volatile flag (in C)

Flagging requires separate put

- Origin PE: `int source[N+1];`

```
initialize_data(source, N)
```

```
source[N] = 1
```

```
put(target, source, N+1, remote_pe)
```

Try to put flag at end of data

- Remote PE: `int target[N+1];`

```
// assume previous initialisation target[N] = -1
```

```
shmem_wait(target[N], -1)
```

Send data and flag together

- **Incorrect!**

- no guarantee of order of data arrival
- even *within* a single put call

Assume arrival of flag means arrival of data

Collectives

- Many collective patterns recur in parallel codes
 - broadcast
 - global sum
 - ...
- OpenSHMEM provides higher-level routines
 - analogous to MPI collectives ...
 - ... but harder to use!
- Issues
 - user must provide (and maybe initialise) various workspace buffers
 - only certain subsets can be specified
 - synchronisation issues between calls

Example: global sum of double

```
void shmem_double_sum_to_all(double *target, double *source,  
int nreduce, int PE_start, int logPE_stride, int PE_size,  
double *pWrk, long *pSync);
```

- Parameters

- **target**: output buffer (symmetric storage)
- **source**: input buffer (symmetric storage)
- **nreduce**: number of doubles to reduce (i.e. size of source and target)
- **PE_start**, **logPE_stride**, **PE_size**: *active set* of PEs taking part
- **pWrk**: symmetric work array whose size depends on **nreduce**
- **pSync**: fixed-size symmetric array for synchronisation flags etc.

Notes

- Active sets
 - all PEs in the active set must call the collective routine
 - $\text{start}, \text{start}+2^{\text{stride}}, \text{start} + 2*2^{\text{stride}}, \text{start}+3*2^{\text{stride}}, \dots, \text{start}+(\text{size}-1)*2^{\text{stride}}$
 - the triplet $(0, 0, \text{shmem_n_pes}())$ specifies all the PEs
 - the triplet $(1, 1, \text{shmem_n_pes}()/2)$ specifies all the odd PEs
 - more restrictive than MPI communicators
- Work arrays
 - `pWrk` of size $\text{max}(\text{nreduce}/2+1, \text{_SHMEM_REDUCE_MIN_WRKDATA_SIZE})$
 - in Fortran: $\text{max}(\text{nreduce}/2+1, \text{SHMEM_REDUCE_MIN_WRKDATA_SIZE})$
 - `pSync` of size `_SHMEM_REDUCE_SYNC_SIZE`
 - in Fortran: `SHMEM_REDUCE_SYNC_SIZE`

Collective synchronisation issues

- **pSync** must be initialised prior to *first* call
 - `SHMEM_SYNC_VALUE` (Fortran)
 - `_SHMEM_SYNC_VALUE` (C)
 - may require synchronisation between initialisation and first call
 - values are reset after the call completes
 - or use static initialisation
- Cannot use the same work or sync arrays if two calls can overlap
 - separate by barrier
 - toggle between `pWrk1` and `pWrk2` etc.

Example

```
shmem_double_sum_to_all(xsum, x, 1, 0, 0, shmem_n_pes(),  
                        pWrk, pSync);
```

```
// Ensure reduction is over before reusing workspace
```

```
shmem_barrier_all();
```

```
shmem_double_sum_to_all(ysum, y, 1, 0, 0, shmem_n_pes(),  
                        pWrk, pSync);
```

```
...
```

```
shmem_double_sum_to_all(xsum, x, 1, 0, 0, shmem_n_pes(),  
                        pWrk1, pSync1);
```

```
// Use different workspace for next reduction
```

```
shmem_double_sum_to_all(ysum, y, 1, 0, 0, shmem_n_pes(),  
                        pWrk2, pSync2);
```

Strided transfers

- Simple strided patterns can be sent in a single put
 - more restrictive than even `MPI_Type_vector()`

```
double precision, save :: x(0:N+1, 0:N+1)
// send halo up in the 2nd dimension
CALL SHMEM_DOUBLE_IPUT(x(0,1), x(N+1,1) N+2, N+2, N, pe_up)
```

- Sends N data elements separated by $N+2$
 - here it picks out $x(N+1, 1)$, $x(N+1, 2)$, ..., $x(N+1, N)$ at source
 - writes to $x(0, 1)$, $x(0, 2)$, ..., $x(0, N)$ at target on `pe_up`
- Can specify different strides at target and source

Dynamic memory allocation (C)

- Static allocation in symmetric memory is very restrictive

- In C, use an alternative to malloc

- `void *shmalloc(size_t size);`

```
// allocate reduction workspace
```

```
double *pWrk;
```

```
pWrksize = max(nreduce/2+1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE);
```

```
pWrk = (double *) shmalloc(pWrksize*sizeof(double));
```

- Must be called by all PEs (a collective routine)
 - Usual issues with C multidimensional arrays, e.g. see `dosharpen.c`
 - also have `shfree()`;

Dynamic memory allocation (Fortran)

- Malloc-like routine provided in Fortran
 - `CALL SHPALLOC (addr, length, errcode, abort)`
 - `addr` is a “Cray pointer” to an array; `length` counted in **32-bit words**
 - last two arguments relate to behaviour on error (see manual)

- Relatively simple for 1D arrays

array contains 64-bit doubles

```
double precision :: pWrk(1) ! Dummy declaration
pointer (addr, pWrk) ! Get pointer to array
call shpalloc(addr, 2*pWrksize, errcode, 0)
pWrk(3) = 99
```


Multidimensional Fortran arrays

- Compiler needs to know leading array dimensions
 - cannot just declare dimensions as 1

```
double precision :: matrix(N,N) ! Dummy declaration
pointer (maddr, matrix)          ! Get pointer
...
! before shpalloc, no storage associated with matrix
call shpalloc(maddr, 2*N*N, errcode, 0)
matrix(7,4) = 34.0
```

- see `dosharpen.f90` for real examples
- Also have `shpdeal1c()`

Locks

- Can lock integer variables
 - this is a global lock (e.g. stored on PE 0) which could be used for critical sections etc.

```
shmem_set_lock(lock);  
shmem_clear_lock(lock);  
islocked = shmem_test_lock(lock);
```

- all locks must be initialised to zero
- Can be used to protect access to data
 - requires all code to respect association of lock with data

Atomic Memory Operations

- Locks can be very heavyweight for simple operations
 - e.g. adding one to a remote variable:

```
get pointer for lock on remote pe
obtain the lock
get value from remote pe
add one to value
put value back
release lock
```
- OpenSHMEM has atomic memory operations
 - e.g., `CALL SHMEM_INT4_ADD(target, value, remote_pe)`
 - atomically adds `value` to `target` on `remote_pe`
 - also have increment, swap, fetch-and-add,...

Summary

- OpenSHMEM contains all the routines you would expect of a PGAS library
- A bit confusing in places, often due to history of non-standard implementations
- May be more portable than languages such as UPC and coarrays
 - does not require compiler support
- Very efficient on Cray platforms