OpenSHMEM

http://www.openshmem.org/site/

George Bosilca

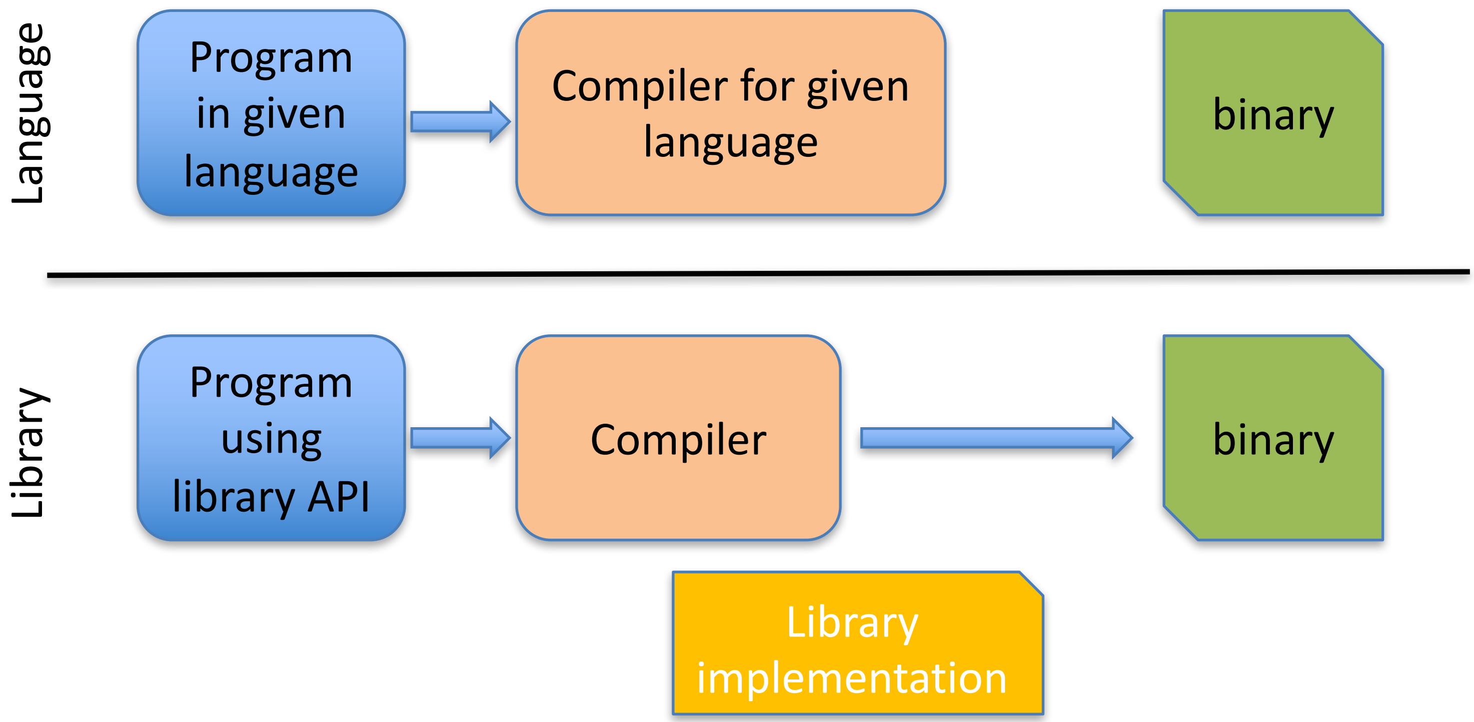
CS462 – Fall 2016

Parallel Programming Models

- What exactly means parallel: An extension to concurrency where things happens on different locations (processors)
- One simple way to differentiate programming models is by their address space: global vs. distributed
  - Global: the address space is reachable by every process (think threading or OpenMP)
  - Distributed: each process address space is private, access only goes through specialized API (MPI)
  - Middle ground: partitioned global address (PGAS descendants) where some parts are private and some shared

The PGAS family

- Libraries: GASNet, ARMCI / Global Arrays, GASPI/GPI, OpenSHMEM
- Languages: Chapel, Titanuim, X10, UPC, CoArray Fortran



PGAS Languages vs Libraries

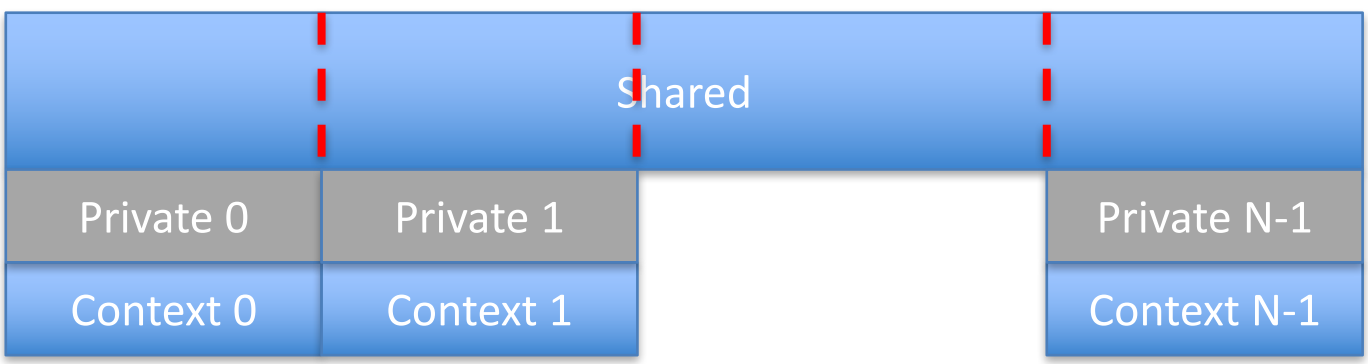
Languages	Libraries
Often more concise	More information redundancy in program
Requires compiler support	Generally not dependent on a particular compiler
More compiler optimization opportunities	Library calls are a "black box" to compiler, typically inhibiting optimization
User may have less control over performance	Often usable from many different languages through bindings
Examples: UPC, CAF, Titanium, Chapel, X10	**Examples:** OpenSHMEM, Global Arrays, MPI-3

UNIVERSITY of HOUSTON

Courtesy Dr. Barbara Chapman

PGAS

- Execution entities share a common shared memory region distributed among all participants



Unified Parallel C (UPC)

- Language defines a “physical” association between execution contexts (UPC threads) and shared data items called “affinity”
  - Scalars data is affine with execution context 0
  - Standard data distribution concepts applies: cyclic, block and block-cyclic
- All interactions with shared data explicitly managed by the application developer.
  - UPC provides a toolbox of basic primitives: locks, barriers, fences.
- Load balancing is done using the forall concept

CoArray Fortran

- SPMD-like: multiple images, each with it's own index (similar to rank in MPI), exists
- Each image execute independently of the others … but the same program
- Synchronizations between images is explicit
- An "object" (data) has the same name in all images
- An image can only work in local data
- An image moves remote data to local data, using explicit CAF syntax.
- No data movement outside this concept is allowed.

Symmetrical Hierarchical MEMory

- SPMD application developed in C, C++ and Fortran
- Similar to CAF: programs perform computations in their own address space but
  - Explicitly communicate data and synchronize with the other processes
- A process participating in SHMEM applications are called processing elements (PE)
- SHMEM provides remote one-sided data transfer, some basic collective concepts (broadcast and reduction), specialized synchronizations and atomic memory operation (remote memory)

History of SHMEM

- Originator: similar time-frame as MPI
  - SHMEM in 1993 by Cray Research (for Cray T3D)
  - SGI incorporated Cray SHMEM in their Message Passing Toolkit (MPT)
  - Quadrics optimized it for QsNet. First come to the Linux world
  - Many others: GSHMEM, University of Florida; HP, IBM, GPSHMEM (ARMCI).
- Unlike MPI, SHMEM was not defined by a standard. A loose API was used instead..
  - In other words, while all implementations manipulated similar concepts they were all different.
  - A push for standardization was necessary (OpenSHMEM)

OpenSHMEM

- An effort to create a standardized SHMEM library API with a [clear] well-defined behavior
- SGI SHMEM API is the baseline for OpenSHMEM 1.0
- A forum to discuss and extend the SHMEM standard with critical new capabilities
  - http://openshmem.org/site/
  - As of September 2016 the Open SHMEM standard reached version 1.3

Everything evolves around

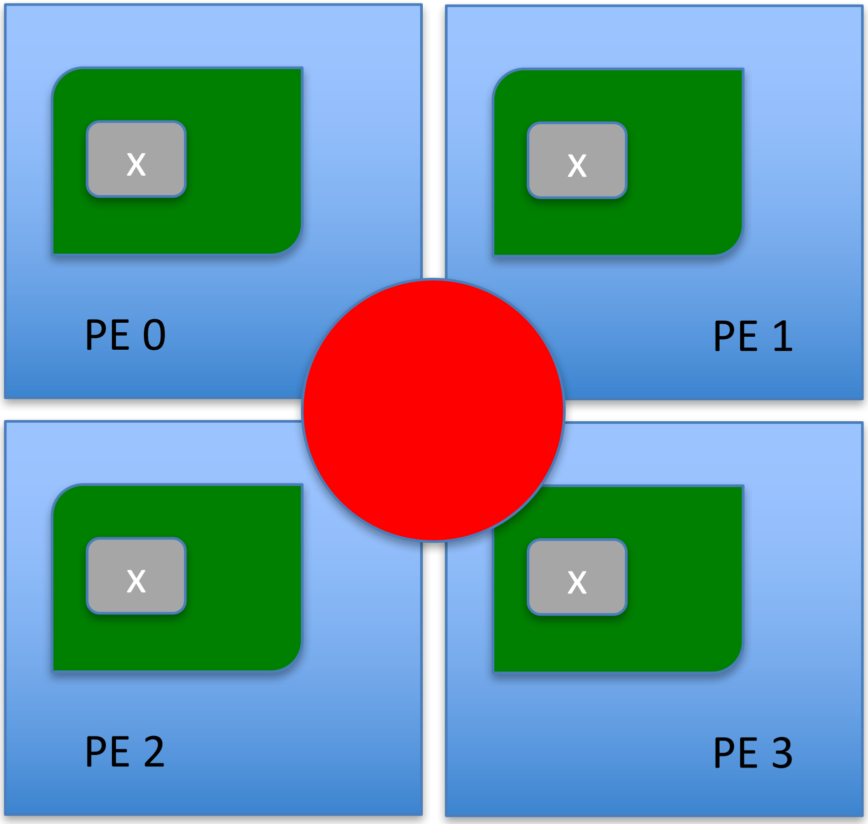
- Remote Direct Memory Access (RDMA)
  - RDMA allows one PE to access certain variables of another PE without interrupting the other PE
  - These data transfers are completely asynchronous
  - They can take advantage of hardware support
- Terminology
  - PE: processing element, a numbered process
  - Origin: process that performs the call
  - Remote_pe: process on each the memory is accessed
  - Source: array which the data is copied from
  - Target: array which the data is copied to
- The key concept here is the symmetric variables
  - Force the applications to be SPMD

Symmetric Variables

- Scalars or arrays that exists with the same size, type, and relative address on all PEs.
- They can either be
  - Global (static variables, or local variables)
  - Dynamically allocated and maintained by the SHMEM library
- With little help from the Operating System, the following types of objects can be made symmetric:
  - Fortran data objects: common blocks and SAVE attributes
  - Non-stack C and C++ variables
  - Fortran arrays allocated with shpalloc
  - C and C++ data allocated by shmalloc

Example (dynamic allocation)

```
int main (void)
{
  int *x;
 …
  start_pes(4);
 …
  x = (int*) shmalloc(sizeof(x));
 …
  shmem_barrier_all();
 …
  shfree(x);
  return 0;
}
```



OpenSHMEM primitives

- Initialization and Query
- Symmetric Data Management
- Data transfers: puts and gets (RDMA)
- Synchronization: barrier, fence, quiet
- Collective: broadcast, collection (allgather), reduction
- Atomic Memory Operations
  - Mutual Exclusion
  - Swap, add, increment, fetch
- Distributed Locks
  - Set, free and query
- Accessibility Query Routines
  - PE accessible, Data accessible

Main Concept

- As the data transfers are one-sided, it is difficult to maintain a consistent view of the state of the parallel application
  - Only local completion is known, and only in some cases
  - Example: put operation
- Synchronization primitives should be used to enforce completion of communication steps

Initialization and Query

- **void** start_pes(**int** npes);
- **int** shmem_my_pe(**void**);
- **int** shmem_n_pes(**void**);
- **int** shmem_pe_accessible(**int** pe);
- **int** shmem_addr_accessible(**void** *addr, **int** pe);
- **void** *shmem_ptr(**void** *target, **int** pe);
  - Only if the target process is running from the same executable (symmetry of the global variables)

Your first OpenSHMEM application

```
#include <stdio.h>
#include <shmem.h> /* The shmem header file */
int
main (int argc, char *argv[])
{
   int nprocs, me;
   start_pes (4);
   nprocs = shmem_n_pes (); me = shmem_my_pe ();
   printf ("Hello from %d of %d\n", me, nprocs); return 0;
}
```

```
Hello from 0 of 4
Hello from 2 of 4
Hello from 3 of 4
Hello from 1 of 4
```

Symmetric Data Management

- Allocate symmetric, remotely accessible blocks (the call are extremely similar to their POSIX counterpart)
  - void *shmalloc(size_t size);
  - void shfree(void *ptr);
  - void *shrealloc(void *ptr, size_t size);
  - void *shmemalign(size_t alignment, size_t size);
  - extern long malloc_error;
- These calls are collective, which means all processes involved in the execution must make them
  - This is a simple way to ensure the symmetry of all dynamically allocated variables

Remote Memory Access - PUT

- void shmem_<type>_p(<type>* target,
  <type> value, int pe);
  void shmem_<type>_put(<type>* target,
  const <type> *source, size_t len, int pe);

- Type can be: floating point [double, float], integer [short, int, long, longdouble, longlong]
- void shmem_putXX(void *target,
  const void *source, size_t len, int pe);
- XX can be: 32, 64, 128
- void shmem_putmem(void *target,
  const void *source, size_t len, int pe);
  - Byte level function

Remote Memory Access - PUT

- Moves data from local memory to remote memory:
  - Target: remotely accessible object where the data will be moved
  - Source: local data object containing the data to be copied
  - Len: number of elements in the source (and target) array. The type of elements (from the function name) will decide how much data will be transferred
  - Pe: the target PE for the operation
- If there is only one data to copy there is an alias shmem_<type>_p

Example - PUT

```
..
long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
static long target[10];

start_pes(2);

if ( _my_pe() == 0 ) {
  /* put 10 words into target on PE 1 */
  shmem_long_put(target, source, 10, 1);
}
shmem_barrier_all(); /* sync sender and receiver */

if (_my_pe() == 1) {
  for( i = 0; i < 10; i++ )
    printf("target[i] on PE %d is %d\n", i, _my_pe(), target[i]);
}
...
```

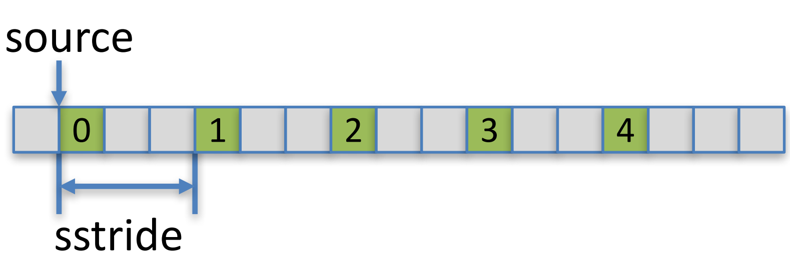
Target should be in a symmetric memory

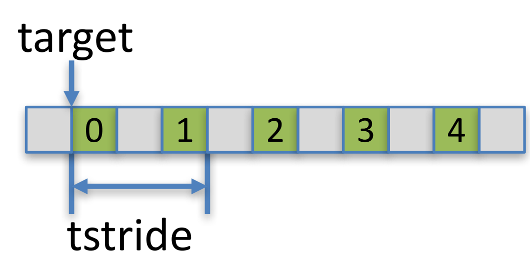
Without synchronization the target PE does not know when the data is available

No assumption about the order of operations should be made

Remote Memory Access - IPUT

- void shmem_<TYPE>_iput(<TYPE> *target,
  const <TYPE> *source, ptrdiff_t tstride,
  ptrdiff_t sstride, size_t nelems, int pe);
- Same idea as PUT plus
  - tstride: the stride between elements on the target array
  - sstride: the stride between elements on the source array





Remote Memory Access - GET

- <type> shmem_<type>_g(<type>* target, int pe);
  void shmem_<type>_get(<type>* target,
  const <type> *source, size_t len, int pe);

- Type can be: floating point [double, float], integer [short, int, long, longdouble, longlong]
- void shmem_getXX(void *target,
  const void *source, size_t len, int pe);
- XX can be: 32, 64, 128
- void shmem_getmem(void *target,
  const void *source, size_t len, int pe);
  - Byte level function

Remote Memory Access - GET

- Moves data from remote memory to local memory:
  - Target: local data object containing the data to be copied
  - Source: remotely accessible object where the data will be moved
  - Len: number of elements in the source (and target) array. The type of elements (from the function name) will decide how much data will be transferred
  - Pe: the source PE for the operation
- If there is only one data to copy there is an alias shmem_<type>_g

Example - GET

```
..
long source;
static long target[10];

start_pes(2);
source = _my_pe();

if ( _my_pe() == 0 ) {
  /* get 1 words from each target PE */
  for( t = 0; t <  _num_pe(); t++)
    shmem_long_get(target + t, &source, 1, t);
}
shmem_barrier_all(); /* sync sender and receiver */

if ( _my_pe() == 0 ) {
  for( i = 0; I < _num_pe(); i++ )
    printf("target[%d] on PE %d is %d\n", i, target[i], target[i]);
}
…
```

Target should be in a symmetric memory

No need for synchronization after the call. The call is blocking it returns once the operation is completed

Consecutive gets complete in order

Example - GET

```
..
long source;
static long target[10];

start_pes(2);
source = _my_pe();

if ( _my_pe() == 0 ) {
  /* get 1 words from ea[illegible]
 for( t = 0; t < [illegible]
   shmem[illegible], 1, t);
}
[illegible]sync sender and receiver */

[illegible]_pe() == 0 ) {
 for( i = 0; I < _num_pe(); i++ )
   printf("target[%d] on PE %d is %d\n", i, target[i], target[i]);
}
...
```

This example is WRONG !!!

Example - GET

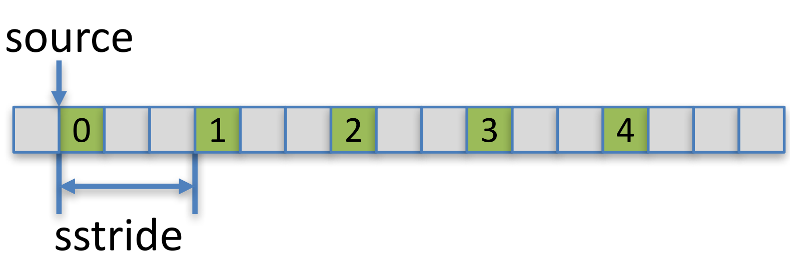
```
..
long source;
static long target[10];

start_pes(2);
source = _my_pe();
shmem_barrier_all(); /* sync sender and receiver */
if ( _my_pe() == 0 ) {
  /* get 1 words from each target PE */
 for( t = 0; t < _num_pe(); t++)
    shmem_long_get(target + t, &source, 1, t);
}
shmem_barrier_all(); /* sync sender and receiver */

if ( _my_pe() == 0 ) {
  for( i = 0; I < _num_pe(); i++ )
    printf("target[%d] on PE %d is %d\n", i, target[i], target[i]);
}
…
```

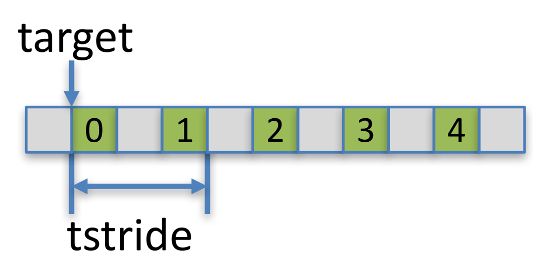
This barrier is needed to ensure proper initialization for source on all Pes.

We need

Remote Memory Access - IGET

- void shmem_<TYPE>_iget(<TYPE> *target, const <TYPE> *source, ptrdiff_t tstride, ptrdiff_t sstride, size_t nelems, int pe);
- Expand the capabilities of GET with
  - tstride: the stride between elements on the target array
  - sstride: the stride between elements on the source array





Remote Memory Access

- Put vs. Get
  - Put call completes when data is "being sent"
  - Get call completes when data is "stored locally"
- Cannot assume put has written until later synchronization
  - Data still in transit
  - Partially written at target
  - Put order changed by e.g. network
- Puts allow overlap
  - Communicate / Compute / Synchronize