OpenSHMEM

George Bosilca

CS594 – Spring 2014
Programming Models

• Data Parallel Model
  – HPF

• Communication Centric Model (Message Passing)
  – MPI

• Shared Memory model
  – OpenMP

• Distributed Shared Memory Model (Partitioned Global Address Space model)
  – UPC, CAF (CoArray Fortran), SHMEM
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 these versions 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
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

• The key concept here is the symmetric variables
Symmetric Variables

• Scalars or arrays that exists with the same size, type, and relative address on all PEs.

• They can either be
  – Globals (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)

```c
int main (void)
{
    int *x;
    ...
    start_pes(4);
    ...
    x = (int*) shmalloc(sizeof(x));
    ...
    shmemBarrier_all();
    ...
    shfree(x);
    return 0;
}
```
OpenSHMEM primitives

• 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
Setup and Query Functions

- **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

```c
#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
```
Memory Management

• Allocate symmetric, remotely accessible blocks
• `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;`
Remote Memory Access - PUT

• void shmem_<type>_put(<type>* target, const <type>* source, size_t len, int pe);
• Type can be: double, float, 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[0] on PE %d is %d\n", _my_pe(), target[0]);
}
...

...
Remote Memory Access - IPUT

• void shmem_<TYPE>_iput(<TYPE> *target, const <TYPE> *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
• Same idea as PUT plus
  – Tst: the stride between elements on the target array
  – Sst: the stride between elements on the source array
Remote Memory Access - GET

- void shmem_<type>_get(<type>** target,
  const <type>** source, size_t len, int pe);
- Type can be: double, float, 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_getmemmem(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 = 1; t < _num_pe(); t++)
        shmemp_long_get(target + t, source, 1, t);
}
shmemp_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[0]);
}
...

Example - GET

..

long source;
static long target[10];

start_pes(0);
source = _my_pe();
if ( (_my_pe() == 0) {
    /* get 1 words from each target*/
    for( t = 1; t < _num_pe();
        shmem_long
    }
    /* sync sender and receiver */
    for( i = 0;i<_num_pe();i++)
        printf("target[%d] on PE %d is %d\n", i, target[0]);
}
...

This example is WRONG !!!
Example - GET

..

long source;
static long target[10];

start_pes(0);
source = _my_pe();
shmem_barrier_all(); /* sync sender and receiver */
if ( (_my_pe() == 0) {
    /* get 1 words from each target PE */
    for( t = 1; 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[0]);
}
...

Remote Memory Access - IGET

- `void shmem_<TYPE>_iget(<TYPE> *target, const <TYPE> *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
- Same idea as GETplus
  - Tst: the stride between elements on the target array
  - Sst: the stride between elements on the source array
Atomic Memory Operations (AMO)

• One-sided mechanism that combines memory update operations with atomicity guarantee

• Two types of AMO routines:
  – Non-fetch: update the remote memory in a single atomic operation. No completion is imposed as there is no local return value related to the operation.
  – Fetch-and-operate: combine memory update and fetch operations in a single atomic operation. The routines return after the data has been fetched and locally delivered.
AMO: fetch: CSWAP

• <type> shmem_<type>_cswap(<type>* target, <type> cond, <type> value, int pe);
  – type: int, long, longlong
  – The function returns the old value of *target
  – Target: remotely accessible integer data object to be updated
  – Cond: the value to be compared with. If the remote target and the cond value are equal, then value is swapped into the remote target. Otherwise, the remote target is unchanged.
AMO: fetch: SWAP

• `<type> shmem_<type>_swap(<type>* target, <type>value, int pe);`
  – type: float, double, int, long, longlong
  – The function returns the old value of *target
  – Target: remotely accessible integer data object to be updated
  – the remote target is swaped with value into the remote target
AMO: fetch: FINC, FADD

- `<type> shmem_<type>_finc(<type> *target, int pe);`
- `<type> shmem_<type>_fadd(<type> *target, <type> value, int pe);`
  - Atomic fetch-and-increment/add on the remote data object with 1/value
  - Returns the previous value in *target
AMO: non-fetch: INC, ADD

- void shmem_<type>_inc(<type> *target, int pe);
- void shmem_<type>_add(<type> *target, <type> value, int pe);
  - Atomic increment/add on the remote data object with 1/value
  - Returns ... nothing
Collective Operations: Barrier_all

• void shmem_barrier_all(void)
  – Barrier between all PE. All operations issued before the barrier are completed upon return.
  – This operation complete al remote shmem_<type>_add and put.
Collective Operations: Barrier

• void shmem_barrier(int PE_start, int logPE_stride, int PE_size, long *pSync)

• Define a barrier on a log (base 2) group of PE

• pSync: must be a symmetric array of type long, that is dedicated for the operation (of size __SHMEM_BARRIER_SYNC_SIZE). Upon entry it must contain __SHMEM_SYNC_VALUE. Upon return it will contain the same value.

• pSync is used internally for coordination and should not be modified during the operation on any PE.
#include <stdio.h>
#include <shmem.h>

long pSync[_SHMEM_BARRIER_SYNC_SIZE];
int x = 10101;

int main(void)
{
    int me, npes;
    for (int i = 0; i < _SHMEM_BARRIER_SYNC_SIZE; i += 1){
        pSync[i] = _SHMEM_SYNC_VALUE;
    }

    start_pes(0);
    me = _my_pe();
    npes = _num_pes();
    if(me % 2 == 0) {
        x = 1000 + me;
        /*put to next even PE in a circular fashion*/
        shmем_int_p(&x, 4, me+2%npes);
        /*synchronize all even pes*/
        shmем_barrier(0, 1, (npes/2 + npes%2), pSync);
    }

    printf("%d: x = %d\n", me, x);
    return 0;
}
Collective Operations: Broadcast

- void shmem_broadcastXX(void *target, const void *source, size_t nlong, int PE_root, int PE_start, int logPE_stride, int PE_size, long *pSync);
  - XX can be 32 or 64
  - Similar concept to MPI_Bcast: broadcast a block of data from one PE to others PE
  - The participants group is defined by the PE_root, PE_start, logPE_stride and PE_size.
  - The PE_root is a zero-based ordinal with respect to the active set of participants
  - pSync should follow the same rules as for the barrier
Collective Operations: Reductions

- void shmem_<type>_<op>_to_all(
  <type> *target, <type>*source, int nreduce,
  int PE_start, int logPE_stride, int PE_size,
  <type>*pWrk, long *pSync);
  - Type might be: short, int, long, longlong, float, double
  - Op might be: and, or, xor, max, min, sum, prod
Collective Operations: Gather

- void shmem_collectXX(void *target, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
  - XX might be 32 or 64
- Concatenates blocks of data from multiple PEs to an array in every PE (similar to MPI_Allgather)
- The group of participants is defined by the PE_start, logPE_stride and PE_size
- The data is concatenated based on the PE index in the active set.
Point-to-point synchronizations

- void shmem_<type>_wait(<type> *var, int value);
  void shmem_<type>_wait_until(<type> *var, int cond, int value);

- Blocking function waiting until the condition on the *var is true with respect to the value

- The condition can be: equal, not equal, greater than, less or equal than, less than, greater or equal to
Memory Ordering Operations

• As most of the operations are not synchronizing there is a need for enforcing ordering
  – Basically a remote happen-before type of relationship between code blocks
  – void shmem_quiet(void): wait for completion of all outstanding Put, AMO and store operation issues by the PE
  – void shmem_fence(void): assure ordering of delivery of Put, AMO and store operations. All operation prior to the call to shmem_fence are guaranteed to be ordered to be delivered before any subsequent Put, AMO or store operation.
```c
#include <stdio.h>
#include <shmemp.h>

long target[10] = {0};
int targ = 0;
int main(void)
{
    long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
    int src = 99;
    start_pes(0);
    if (!_my_pe() == 0) {
        shmem_long_put(target, source, 10, 1); /*put1*/
        shmem_long_put(target, source, 10, 2); /*put2*/
        shmem_fence();
        shmem_int_put(&targ, &src, 1, 1); /*put3*/
        shmem_int_put(&targ, &src, 1, 2); /*put4*/
    }
    shmem_barrier_all(); /* sync sender and receiver */
    printf("target[0] on PE %d is %d\n", _my_pe(), target[0]);
    return 1;
}
```