OpenSHMEM TUTORIAL

Presenters: Swaroop Pophale and Tony Curtis
University of Houston, Texas
This work was supported by the United States Department of Defense & used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.
Outline

- About us
- Background
- History and Implementations
- The OpenSHMEM Effort
- OpenSHMEM API
- Porting
- A look ahead…
- References
OpenSHMEM Tutorial
Introductory Material

Dr. Barbara Chapman

Tony Curtis

Swaroop Pophale

Ricardo Mauricio

Ram Nanjegowda
OpenSHMEM Tutorial
Introductory Material

- Research in a number of areas
  - Focused on large scale parallelism
  - Exascale
- ~ 20 MS & PhD students
- 3 senior staff and assistant professor
So what is it our group researches?

- OpenMP
  - Extreme scale
  - Distributed systems
  - Locality
Compiler technology
- OpenUH (based on Open64)
Heterogeneous Computing
  - Power-aware
  - OpenMP
  - MCA
  - Accelerators
OpenSHMEM Tutorial
Introductory Material

- PGAS Languages and Libraries
  - UPC
  - CAF
  - Both supported in OpenUH
OpenSHMEM Tutorial
Introductory Material

- PGAS Languages and Libraries
  - SHMEM
  - Chapel
NEW TO PARALLEL COMPUTING?

WE WILL TAKE IT SEQUENTIALLY..
Background

What is Parallel Computing?

- **Parallel computing** is the simultaneous use of multiple compute resources to solve a computational problem.

Sequential Program

```c
main{
    Initialize...
    Compute...
    return 0;
}
```
Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.

Concurrent ≠ Parallel
Background

Different types Parallel Programming

- Single Program Multiple Data (SPMD)
  - All processes are doing the same thing with different data items

- Multiple Program Multiple Data (MPMD)
  - All processes are executing different programs and using different data items
Background

What is a Programming Model?

- A view of data and execution
- Where architecture and applications meet
- Can be viewed as a “contract”
  - Everyone knows the rules
  - Better understanding of performance considerations
- Benefits
  - Application - independence from architecture
  - Architecture - independence from applications
Background
Programming Models

- Data Parallel Model
  - HPF
- Communication Centric Model
  - MPI
- Shared Memory Model
  - OpenMP
- Distributed-Shared Memory Model or the Partitioned Global Address Space Model
  - UPC, CAF, SHMEM
Background

PGAS Programming Model

Logical Layout of PGAS Programming Model
Background

UPC

- Unified Parallel C
- Language defines a "physical" association between shared data items and UPC threads called "affinity".
  - All scalar data has affinity with thread 0.
  - Arrays may have cyclic (per element), blocked-cyclic (user-defined) or blocked (run-time) affinity.
- All thread interaction is explicitly managed by the programmer through language primitives: locks, barriers, memory fences.
- Work sharing using “forall”
The number of images is fixed and each image has its own index, retrievable at run-time.

Each image executes the same program independently of the others.

The programmer inserts explicit synchronization and branching as needed.

An “object” has the same name in each image.

Each image works on its own local data.

An image moves remote data to local data through, and only through, explicit CAF syntax.
Introduction

What is SHMEM?

- Symmetric Hierarchical MEMory library
  - For Single Program Multiple Data style of programming
  - Available for C, C++, and Fortran

- Used for programs that
  - perform computations in separate address spaces and
  - explicitly communicate data to and from different processes in the program.

- The processes participating in SHMEM applications are referred to as processing elements (PEs).

- SHMEM routines supply remote one-sided data transfer, broadcast, reduction, synchronization, and atomic memory operations.
Introduction

History of SHMEM

- **Cray SHMEM**
  - SHMEM first introduced by Cray Research Inc. in 1993 for Cray T3D
  - Platforms: Cray T3D, T3E, PVP, XT series

- **SGI SHMEM**
  - SGI incorporated Cray SHMEM in their Message Passing Toolkit (MPT)
  - Owns the “rights” for SHMEM

- **Quadrics SHMEM** (company out of business)
  - Optimized API for QsNet
  - Platform: Linux cluster with QsNet interconnect

- **Others**
  - GSHMEM, University of Florida
  - HP SHMEM, IBM SHMEM (used internally only)
  - GPSHMEM (cluster with ARMCI & MPI support, dead)

Note: SHMEM was not defined by any one standard.
The Problem: Differences in SHMEM Implementations (1)

- Initialization
  - Include header shmem.h
    - E.g. `#include <shmem.h>`, `#include <mpp/shmem.h>`
  - `start_pes, shmem_init`: Initializes the library
  - `my_pe`: Get the PE ID of local processor (0 to N-1)
  - `num_pes`: Get the total number of PEs in the program

<table>
<thead>
<tr>
<th></th>
<th>SGI</th>
<th>Quadrics</th>
<th>Cray</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fortran</td>
<td>C/C++</td>
<td>C/C++</td>
</tr>
<tr>
<td>start_pes(0)</td>
<td>start_pes(0)</td>
<td>shmем_init</td>
<td>start_pes</td>
</tr>
<tr>
<td>NUM_PES</td>
<td></td>
<td>num_pe</td>
<td>num_pe</td>
</tr>
<tr>
<td>MY_PE</td>
<td></td>
<td>my_pe</td>
<td>my_pe</td>
</tr>
</tbody>
</table>
The Problem: Differences in SHMEM Implementations (2)

Hello World (SGI on Altix)

#include <stdio.h>
#include <mpp/shmem.h>

int main(void)
{
    int me, npes;

    start_pes(0);
    npes = _num_pes();
    me = _my_pe();
    printf("Hello from %d of %d\n", me, npes);
    return 0;
}

Hello World (SiCortex)

#include <stdio.h>
#include <shmem.h>

int main(void)
{
    int me, npes;

    shmem_init();
    npes = num_pes();
    me = my_pe();
    printf("Hello from %d of %d\n", me, npes);
    return 0;
}
The Problem:
Differences in SHMEM Implementations (2)

Hello World on SGI on Altix

```c
#include <stdio.h>
#include <mpp/shmem.h>
int main(void)
{
    int me, npes;
    start_pes(0);
    npes = _num_pes();
    me = _my_pe();
    printf("Hello from %d of %d\n", me, npes);
    return 0;
}
```

Hello World on SiCortex

```c
#include <stdio.h>
#include <shmem.h>
int main(void)
{
    int me, npes;
    shmem_init();
    npes = num_pes();
    me = my_pe();
    printf("Hello from %d of %d\n", me, npes);
    return 0;
}
```
The Solution:

OpenSHMEM
What is OpenSHMEM?

- An effort to create a standardized SHMEM library API and defining expected behavior
- Aims at bringing together hardware vendors and SHMEM library developers
- Discuss and extend standard with important new capabilities

SGI’s SHMEM API is the baseline for OpenSHMEM Specification 1.0
OpenSHMEM Outreach

- Community web site (under construction)
- Wiki
- Documentation
  - OpenSHMEM 1.0 Specification
  - FAQ
  - Cheat sheet
- Training material and tutorials
- Mailing list
  - [https://email.ornl.gov/mailman/listinfo/openshmem](https://email.ornl.gov/mailman/listinfo/openshmem)
OpenSHMEM Participation

- **PGAS’10**
  - Workshop and paper

- **SC’10 New Orleans**
  - Booth presence (PGAS, Oak Ridge National Laboratory, Gulf Coast Academic Supercomputing)
  - BOF Session
  - GCAS booth presentation

- **ICS 2011**
  - Poster Presentation
OpenSHMEM Participation

- **PGAS’11**
  - Workshop
- **SC’11**
  - Poster
  - BOF
Key Concept
Remote Direct Memory Access

- RDMA lets one PE access certain variables of another PE without interrupting the other PE

- SHMEM can take advantage of hardware RDMA

- SHMEM’s data transfer uses symmetric variables
Key Concept
Symmetric Variables

- Symmetric Variables
  - Scalars or arrays that exist with the same size, type, and relative address on all PEs.

- There are two types of Symmetric Variables
  - Globals
    - Dynamically allocated and maintained by the SHMEM library

- The following kinds of data objects are symmetric:
  - Fortran data objects
    - in common blocks
    - or with the SAVE attribute.
  - Non-stack C and C++ variables.
  - Fortran arrays allocated with shpalloc.
  - C and C++ data allocated by shmalloc.
int main (void)
{
    int *x;
    ...
    start_pes(0);
    ...
    x = (int*) shmalloc(sizeof(x));
    ...
    ...
    shmem_barrier_all();
    ...
    shfree(x);
    return 0;
}
OpenSHMEM Routines

- **Data transfers**
  - One sided puts and gets

- **Synchronization**
  - Barrier, Fence, quiet

- **Collective communication**
  - Broadcast, Collection, Reduction

- **Address Manipulation and Data Cache control**
  - Not supported by all SHMEM implementations (Deprecated in OpenSHMEM 1.0)

- **Atomic Memory Operations**
  - Provide mechanisms to implement mutual exclusion
    - Swap, Add, Increment, fetch

- **Distributed Locks**
  - Set, free and query

- **Accessibility Query Routines**
  - PE accessible, Data accessible
OpenSHMEM API

Data Transfer (1)

- **Put**
  - **Single value**
    - double, float, int, long, short, longlong, longdouble, char
  - **Contiguous object**
    - For C: TYPE = double, float, int, long, longdouble, longlong, short, 32, 64, 128, mem
    - For Fortran: TYPE=complex, integer, real, character, logical
  - **Strided**
    - For C: TYPE = double, float, int, long, longdouble, longlong, short, 32, 64, 128, mem
    - For Fortran: TYPE=complex, integer, real, character, logical
Code snippet showing a put from PE 0 to PE 1

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

start_pes(0);
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]);
}
...
OpenSHMEM API

Data Transfer (3): Put

Excuse me while I overwrite your copy of source

Output

target[0] on PE 1 is 1
target[1] on PE 1 is 2
target[2] on PE 1 is 3
target[3] on PE 1 is 4
...
target[9] on PE 1 is 10
OpenSHMEM API
Data Transfer (4)

- Get
  - Single value
    - double, float, int, long, short, longlong, longdouble, char
  - Contiguous object
    - For C: TYPE = double, float, int, long, longdouble, longlong, short, 32, 64, 128, mem
    - For Fortran: TYPE=complex, integer, real, character, logical
  - Strided
    - For C: TYPE = double, float, int, long, longdouble, longlong, short, 32, 64, 128, mem
    - For Fortran: TYPE=complex, integer, real, character, logical
static long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
long target[10];

start_pes(0);
if (_my_pe() == 1) {
    /* get 10 words into target from PE 0 */
    shmemp_long_get(target, source, 10, 0);
}

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

...
OpenSHMEM API

Data Transfer (5): Get

Excuse me while I get a copy of source

Target[0] on PE 1 is 1
Target[1] on PE 1 is 2
Target[2] on PE 1 is 3
Target[3] on PE 1 is 4
...
Target[9] on PE 1 is 10
OpenSHMEM Collective API
Group Synchronization

- **Barrier**
  - `pSync` is a symmetric work array that enables overlapping collective communication
  - `void shmem_barrier_all()`
    - All PEs wait until every PE calls this function

  **NEW CONCEPT**
  **“ACTIVE SET”**
  Subset of PEs defined by `Start_PE`, `logPE_stride` and `PE_sizes`
Quick look at Active Sets

Example 1
- PE\_start = 0, logPE\_stride = 0, PE\_size = 4
  Active Set? PE 0, PE 1, PE 2, PE 3

Example 2
- PE\_start = 0, logPE\_stride = 1, PE\_size = 4
  Active Set? PE 0, PE 2, PE 4, PE 6

Example 3
- PE\_start = 2, logPE\_stride = 2, PE\_size = 3
  Active Set? PE 2, PE 6, PE 10
Group Synchronization (1): 

shmem_barrier_all()

shmem_barrier_all() synchronizes all executing PEs

Ensures completion of all
- local memory stores
- remote memory updates
Group Synchronization (2): `shmem_barrier_all()`
Group Synchronization (3):

shmem_barrier(...)
Point-to-Point synchronization

- Wait
- Wait Until
  - Equal, Not equal, Greater than, Less than or equal to, Less than, Greater than or equal to

- For C: TYPE = double, float, int, long, longdouble, longlong, short
- For Fortran: TYPE=complex, integer, real, character, logical
OpenSHMEM API
Point-to-Point Synchronization (2)

```c
long *dest;
dest = (long *) shmalloc( sizeof(*dest) );
*dest = 9L;
shmem_barrier_all();
....
if (me == 1) {
    shmem_long_wait(dest, 9L);
}
....
if (me == 0) {
    src=101;
    shmem_long_put(dest, &src, 1, 1);
}
shmem_barrier_all();
...
```

Code snippet showing operation of shmem_wait
OpenSHMEM API

Point-to-Point Synchronization (3)

- **Fence (data transfer sync.)**
  - Ensures ordering of outgoing write (put) operations to a single PE
  - `void shmem_fence()`

- **Quiet (data transfer sync.)**
  - Waits for completion of all outstanding remote writes initiated from the calling PE (on some implementations fence = quiet)
  - `void shmem_quiet()`
OpenSHMEM Collective API

Broadcast (1)

- One-to-all communication
  - void shmem_broadcastSS(void *target, void *source, int nelems, int PE_root, int PE_start, int PE_stride, int PE_size, long *pSync)
  - Storage Size (SS, bits) = 32/4, 64/8
OpenSHMEM Collective API
Broadcast (2)

Code snippet showing operation of `shmem_broadcast`

```c
... int *target, *source; target= (int *) shmalloc( sizeof(int) );
source= (int *) shmalloc( sizeof(int) );
*target= 0;
*source= 101;
shmem_barrier_all();
if (me == 1) {
    *source = 222;
}
shmem_broadcast32(target, source, 1, 0, 0, 0, 4, pSync);
printf("target on PE %d is %d\n", _my_pe(), *target);
...```

Output

target on PE 0 is 0
target on PE 1 is 222
target on PE 2 is 222
target on PE 3 is 222
OpenSHMEM Collective API

Broadcast (3): Working

PE 0

PE 1

PE 2

PE 3

Shared Address Space

Private Address Space
Example 1
- \( PE\_root = 0, \ PE\_start = 0, \ \log PE\_stride = 0, \ PE\_size = 4 \)
- PE 0 broadcasts to PE 1, PE 2 and PE 3

Example 2
- \( PE\_root = 2, \ PE\_start = 2, \ \log PE\_stride = 0, \ PE\_size = 4 \)
- PE 4 broadcasts to PE 2, PE 3 and PE 5

Example 3
- \( PE\_root = 1, \ PE\_start = 0, \ \log PE\_stride = 1, \ PE\_size = 4 \)
- PE 2 broadcasts to PE 0, PE 4 and PE 6
OpenSHMEM Collective API
Collect (1)

- **Collect**
  - Concatenates blocks of data from multiple PEs to an array in every PE
  - `void shmem_collectSS(void *target, void *source, int nelems, int PE_start, int PE_stride, int PE_size, long *pSync)`
  - Storage Size (SS, bits) = 32, 64, 128, mem (any size)

- **Fixed Collect**
  - `void shmem_fcollectSS(void *target, void *source, int nelems, int PE_start, int PE_stride, int PE_size, long *pSync)`
OpenSHMEM Collective API
Collect (2): Working of Collect

PE 0

Shared Address Space
Private Address Space
OpenSHMEM Collective API

Reductions (1)

- Logical
  - and, or, xor
- Extrema
  - max, min
- Arithmetic
  - product, sum
- TYPE = int, long, longlong, short
OpenSHMEM Collective API

Reductions (2): Working

Shared Address Space

Private Address Space
OpenSHMEM API

Atomic Operations (1)

- **Swap**
  - Unconditional
    - `long shmem_swap(long *target, long value, int pe)`
    - `TYPE shmem_TYPE_swap(TYPE *target, TYPE value, int pe)`
      - TYPE = double, float, int, long, longlong, short
  - Conditional
    - `TYPE shmem_TYPE_cswap(TYPE *target, int cond, TYPE value, int pe)`
      - TYPE = int, long, longlong, short

- **Arithmetic**
  - `TYPE shmem_TYPE_OP(TYPE *target, TYPE value, int pe)`
    - OP = fadd, finc
    - TYPE = int, long, longlong, short
... long *dest;
dest = (long *) shmalloc(sizeof(*dest));
*dest = me;
shmem_barrier_all();
....
new_val = me;
if (me == 1) {
    swapped_val = shmem_long_swap(dest, new_val, 0);
    printf("PE %d: dest = %d, swapped = %d\n", me, *target, swapped_val);
}
shmem_barrier_all();
...
OpenSHMEM API

Accessibility

- **shmem_pe_accessible**
  - Determines whether a processing element (PE) is accessible via SHMEM data transfer operations

- **shmem_addr_accessible**
  - Determines whether an address is accessible via SHMEM data transfers operations from the specified remote processing element (PE)
OpenSHMEM API

Mutual Exclusion: Locks

- **Set lock**
  - first-come, first-served manner

- **Clear lock**
  - ensuring that all local and remote stores initiated in the critical region are complete before releasing the lock

- **Test lock**
  - avoid blocking
  - function returns without waiting
OpenSHMEM API

Address Manipulation and Cache

- **Address manipulation**
  - `shmem_ptr` - Returns a pointer to a data object on a remote PE

- **Cache control**
  - `shmem_clear_cache_inv` - Disables automatic cache coherency mode
  - `shmem_set_cache_inv` - Enables automatic cache coherency mode
  - `shmem_set_cache_line_inv` - Enables automatic line cache coherency mode
  - `shmem_udcflush` - Makes the entire user data cache coherent
  - `shmem_udcflush_line` - Makes coherent a cache line
Sequential to Parallel using OpenSHMEM
Parallelization using OpenSHMEM

Step 1

- Preparation
  - Code Analysis
    - To determine what the code does and how it does it
    - Should fit SPMD style of programming
  - Dependency Analysis
    - Data dependencies
      - True dependency, input dependency
    - Control dependencies
      - To determine the sections of code that can run in parallel and those that must be executed sequentially.
Parallelization using OpenSHMEM

Step 2

- Decide what variables need to be symmetric
  - only variables that need to be communicated

- Add shmem API calls for communication and computation

- Add shmem synchronization
  - To insure updates
  - Separate different stages
Parallelization using OpenSHMEM

Example

```c
int main(void)
{
  for(...){
    Parallel
    Compute
    Parallel
    Collect Results
    Synchronize
    collectives
    Synchronize
  }
}
```
Parallelization using OpenSHMEM

Example: SSCA3

- Image Processing and Data I/O Application benchmark developed for High Productivity Computing Systems (HPCS).

- SSCA3 benchmark has essentially two stages;
  - front-end
  - back-end
Parallelization using OpenSHMEM
Example: SSCA3, Our observations

<table>
<thead>
<tr>
<th></th>
<th>UPC</th>
<th>SHMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTACTIC COMPLEXITY</td>
<td>LOW</td>
<td>HIGH</td>
</tr>
<tr>
<td>CONCEPTUAL COMPLEXITY</td>
<td>LOW</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>MAXIMUM SPEED-UP</td>
<td>6.27</td>
<td>8.94</td>
</tr>
</tbody>
</table>
Parallelization using OpenSHMEM

Example: SSCA3
**MPI 1.0 to OpenSHMEM Incremental Porting**

- **Step 1**: Replace initialization calls
- **Step 2**: Replace MPI send-receive pair by a single put/get with appropriate synchronization
- **Step 3**: Replace MPI collective calls with SHMEM collective calls
- **Step 4**: For calls that do not have corresponding OpenSHMEM calls
Example: Stage 1 (Initialization)

```c
#include <mpp/shmem.h>

int main(int argc, char *argv[]){
    start_pes(0);
    comm_size = _num_pes();
    my_rank = _my_pe();
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    ....
    MPI_Finalize();
}```
MPI 1.0 to OpenSHMEM
Incremental Porting: Unmatched calls

MPI_Alltoall( send_count, 1, MPI_INT, recv_count, 1, MPI_INT, MPI_COMM_WORLD );

for(i=0; i<npes; i++){
    shmem_int_put(&recv_count, &send_count, 1, i);
}

Replace by
MPI Code

Distribute blocks of COLUMNS to each process
np = size; // number of processes
blocksize = COLUMNS/np; // block size
B_matrix_displacement = rank * blocksize;

Allocate local arrays
a_local = (double **) malloc(ROWS*sizeof(double *));
b_local = (double **) malloc(ROWS*sizeof(double *));
c_local = (double **) malloc(ROWS*sizeof(double *));

Initialize local arrays
for(i=0; i<ROWS; i++) {
    a_local[i] = (double *) malloc(blocksize*sizeof(double));
    ..
}

OpenSHMEM Code

Distribute blocks to COLUMNS to each process
np = size; // number of processes
blocksize = COLUMNS/np; // block size
B_matrix_displacement = rank * blocksize;

Allocate SHMEM arrays
shmem_barrier_all();
a_local = (double **) shmalloc(ROWS*sizeof(double *));
b_local = (double **) shmalloc(ROWS*sizeof(double *));
c_local = (double **) shmalloc(ROWS*sizeof(double *));

Initialize arrays
for(i=0; i<ROWS; i++) {
    a_local[i] = (double *) shmalloc(blocksize*sizeof(double));
    ..
}
MPI Code

Send the Local block of matrix a to process on right

    MPI_Barrier(MPI_COMM_WORLD);
    if(rank == np-1)
        MPI_Isend (&a_local[i][0],blocksize,MPI_DOUBLE, 0,
                1,MPI_COMM_WORLD,&req[0]);
    else
        MPI_Isend (&a_local[i][0],blocksize,MPI_DOUBLE,rank+1,
                1,MPI_COMM_WORLD,&req[1]);
    if(rank == 0)
        MPI_Recv (&a_local[i][0],blocksize,MPI_DOUBLE,np-1,
                   1,MPI_COMM_WORLD,&status);
    else
        MPI_Recv (&a_local[i][0],blocksize,MPI_DOUBLE,rank-1,
                   1,MPI_COMM_WORLD,&status);

Compute the local displacement...

OpenSHMEM Code

Send the Local block of matrix 'a' to process on right

    shmem_barrier_all();
    if(rank == np-1)
        shmem_double_put(&a_local[i][0],&a_local[i][0],blocksize,0);
    else
        shmem_double_put(&a_local[i][0],&a_local[i][0],blocksize,rank+1);
    shmem_barrier_all();

Compute the local displacement (REMAINS SAME AS MPI)...

...
## MPI 1.0 to OpenSHMEM Direct Replacement (1)

<table>
<thead>
<tr>
<th>MPI calls</th>
<th>Possible OpenSHMEM calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Init( argc, argv )</code></td>
<td><code>start_pes(0)</code></td>
</tr>
<tr>
<td><code>MPI_Comm_rank( MPI_COMM_WORLD, &amp;my_rank )</code></td>
<td><code>_my_pe()</code></td>
</tr>
<tr>
<td><code>MPI_Comm_size( MPI_COMM_WORLD, &amp;comm_size )</code></td>
<td><code>_num_pes()</code></td>
</tr>
<tr>
<td><code>MPI_Barrier(comm)</code></td>
<td><code>shmem_barrier_all()</code></td>
</tr>
<tr>
<td><code>MPI_Allreduce( bucket_size, bucket_size_totals, SIZE, MPI_INT, MPI_SUM, MPI_COMM_WORLD )</code></td>
<td><code>shmem_int_sum_to_all(bucket_size_totals,bucket_size,SIZE,0,0,comm_size,ipWrk,pSync)</code></td>
</tr>
<tr>
<td><code>MPI_Bcast(lt, 1, MPI_INTEGER, 0, MPI_COMM_WORLD)</code></td>
<td><code>shmem_broadcast4(lt, lt, 1, 0, 0, 0, nprocs, pSync)</code></td>
</tr>
</tbody>
</table>
### MPI calls vs. Possible OpenSHMEM calls

<table>
<thead>
<tr>
<th>MPI calls</th>
<th>Possible OpenSHMEM calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Send(send_buff, buff_len, MPI_DOUBLE, to_rank....)</code></td>
<td><code>shmem_double_put(recv_buff, send_buff, buff_len, to_rank)</code></td>
</tr>
<tr>
<td><code>MPI_Recv(recv_buff, buff_len, dp_type, from_rank....)</code></td>
<td><code>shmem_double_get(recv_buff, send_buff, buff_len, from_rank)</code></td>
</tr>
<tr>
<td><code>MPI_Wait(request, status)</code></td>
<td><code>shmem_wait(variable, value)</code></td>
</tr>
<tr>
<td><code>MPI_reduce(t, tmax, 1, MPI_REAL, MPI_MAX, root, mpi_comm_world)</code></td>
<td><code>shmem_int_max_to_all(tmax, t, 1, 0, nprocs, pwrk, psync)</code></td>
</tr>
<tr>
<td><code>MPI_Scatter(src, count, MPI_INT, dst, count, MPI_INT, 0, comm_world)</code></td>
<td><code>shmem_broadcast(dst, src, count, 0, 0, 0, size, pSync)</code></td>
</tr>
<tr>
<td><code>MPI_Gather(src, count, MPI_INT, dst, count, MPI_INT, 0, comm_world)</code></td>
<td><code>shmem_collect32(dst, src, count, 0, 0, 0, size, pSync)</code></td>
</tr>
</tbody>
</table>
### MPI 1.0 to OpenSHMEM Equivalent OpenSHMEM calls

<table>
<thead>
<tr>
<th>MPI calls</th>
<th>Possible OpenSHMEM calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_AlltoAll</td>
<td>for(j1=0;j1&lt;comm_size;j1++){</td>
</tr>
<tr>
<td></td>
<td>shmem_int_put(&amp;recv_count[my_rank], &amp;send_count[j1],1,j1);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>MPI_AlltoAllv</td>
<td>for(j1=0;j1&lt;comm_size;j1++){</td>
</tr>
<tr>
<td></td>
<td>int k1 = send_displ[j1];</td>
</tr>
<tr>
<td></td>
<td>static int k2;</td>
</tr>
<tr>
<td></td>
<td>shmem_int_get(&amp;k2,&amp;recv_displ[my_rank],1,j1);</td>
</tr>
<tr>
<td></td>
<td>shmem_int_put(key_buff2+k2,key_buff1+k1,send_count[j1],j1);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>MPI_Comm and MPI_Group calls</td>
<td>NA</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>NA</td>
</tr>
</tbody>
</table>
OpenSHMEM vs. MPI 2.0
OpenSHMEM vs. MPI 2.0

Symmetric memory allocation

Collective Call

MPI_Win_create(var1, window1)
MPI_Win_create(var2, window2)

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ldots \ldots</td>
<td>\ldots \ldots</td>
</tr>
<tr>
<td>MPI_Put(window1)</td>
<td>MPI_Put(window2)</td>
</tr>
<tr>
<td>\ldots \ldots</td>
<td>\ldots \ldots</td>
</tr>
</tbody>
</table>

MPI Window semantics

- All processes which intend to use the window must participate in window creation.
- Many or all the local allocations/objects should be coalesced within a single window creation.

Symmetric Data

Global variables
Static local or global variables
shmalloc() memory

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_get(&amp;var1)</td>
<td>shmem_put(&amp;var2)</td>
</tr>
<tr>
<td>\ldots \ldots</td>
<td>\ldots \ldots</td>
</tr>
</tbody>
</table>

SHMEM semantics

- All global and static data are by default accessible to all processes.
- Local allocations/objects can be made remotely accessible using shmalloc instead of malloc.
OpenSHMEM vs. MPI 2.0
Synchronization (1)

MPI_Win_fence
• Fence is a collective call.
• Need 2 fence calls, one to separate and another one to complete.
• So it mostly functions like barrier

shmем_fence
• shmем Fence is just meant for ordering of puts.
• It does not separate the processes nor does it mean completion.
• Ensures there are no pending puts to be delivered to the same target before the next put.
OpenSHMEM vs. MPI 2.0
Synchronization (2)

Process 0 (Source)
MPI_Start
……
MPI_Put
MPI_Put
……
MPI_Complete

Process 1 (Dest)
MPI_Post
……
Cannot do anything
……
MPI_Wait

• Point to point synchronization.
• Sender does Start and waits for Post from receiver
• The receiver does Post and waits for the data.
• The sender Puts the data and signals completion to receiver

• The receiver can directly wait for the data using shmem_wait on a event flag.
• The sender puts the data and sets the event flag to signal the receiver.
• Both post and complete are implicit inside the wait and put operation.

Process 0 (Source)
shmem_put(data)
shmem_put(flag1)
shmem_wait(flag2)
……
……

Process 1 (Dest)
shmem_wait(flag1)
……
shmem_put()
shmem_fence()
shmem_put(flag2)
……
### OpenSHMEM vs. MPI 2.0

#### Synchronization (3)

<table>
<thead>
<tr>
<th>Process 0 (Source)</th>
<th>Process 1 (Dest)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if(rank == 0) {</code></td>
<td><code>if(rank == 0) {</code></td>
<td>- No mutual exclusion</td>
</tr>
<tr>
<td><code>MPIWin_lock</code></td>
<td><code>MPIWin_lock</code></td>
<td>- Lock is not real lock, but begin RMA</td>
</tr>
<tr>
<td><code>MPIPut</code></td>
<td><code>MPIPut</code></td>
<td>- Unlock means end RMA</td>
</tr>
<tr>
<td><code>MPIWin_unlock</code></td>
<td><code>MPIWin_unlock</code></td>
<td>- Only the source calls lock</td>
</tr>
</tbody>
</table>

**Process 0 (Source)**

```c
shmem_set_lock
......
shmem_put(data)
......
shmem_clear_lock()
```

**Process 1 (Dest)**

```c
shmem_set_lock
......
shmem_put(data)
......
shmem_clear_lock()
```

- Enforces mutual exclusion
- The PE which acquires lock does put
- The waiting PE gets the lock on first come first served basis
Window creation is a collective operation
May restrict the use of passive-target RMA operations to only work on memory allocated using MPI_Alloc_mem
It is erroneous to have concurrent conflicting RMA get/put (or local load/store)
Multiple windows are allowed to include overlapping memory regions, however it is erroneous to use concurrent operations to distinct overlapping windows
OpenSHMEM and Hardware

- OpenSHMEM is intended to be a specification that
  - Standardizes current efforts
  - Doesn’t restrict implementors

- Want to allow freedom for innovation on hardware
  - E.g. collectives/atomics on NICs
  - Emerging manycore architectures
    - MIC, Bluegene/Q
    - Embedded systems with DMA engines
  - Heterogeneous architectures
    - E.g. Convey, “ceepee-geepee”
References


2. SHMEM tutorial by Hung-Hsun Su, HCS Research Laboratory, University of Florida

3. Evaluating Error Detection Capabilities of UPC Compilers and Runtime Error detection by Iowa State University http://hpcgroup.public.iastate.edu/CTED/


7. Karl Feind, Shared Memory Access (SHMEM) Routines

8. Galen M. Shipman and Stephen W. Poole, Open-SHMEM: Towards a Unified RMA Model
Thanks!

Questions?