OpenSHMEM
Application Programming Interface

http://www.openshmem.org/

Version 1.1

3rd June 2014

Developed by

- High Performance Computing Tools group at the University of Houston
  http://www.cs.uh.edu/~hpctools/

- Extreme Scale Systems Center, Oak Ridge National Laboratory
  http://www.csm.ornl.gov/essc/
Sponsored by
- U.S. Department of Defense (DoD)
  http://www.defense.gov/
- Oak Ridge National Laboratory (ORNL)
  http://www.ornl.gov/

Authors and Collaborators
- Tony Curtis, University of Houston (UH)
- Swaroop Pophale, UH
- Barbara Chapman, UH
- Stephen Poole, ORNL
- Jeff Kuehn, Los Alamos National Laboratory (LANL)
- Oscar Hernandez, ORNL
- Manjunath Gorentla Venkata, ORNL
- Pavel Shamis, ORNL
- Gregory Koenig, ORNL
- Eduardo D’Azevedo, ORNL
- Monika ten Bruggencate, Cray Inc.
- Jens Manser, DoD
- Nick Park, DoD
- Lauren Smith, DoD
- Karl Feind, SGI
- Michael Raymond, SGI

Acknowledgements
The OpenSHMEM specification belongs to Open Source Software Solutions, Inc. (OSSS), a non-profit organization, under an agreement with SGI. The development work of the specification is supported by the Oak Ridge National Laboratory Extreme Scale Systems Center and the Department of Defense.

We would also like to acknowledge the contribution of the members of the OpenSHMEM mailing list for their ideas, discussions, suggestions, and constructive criticism which has helped us improve this document.
## Contents

1. The OpenSHMEM Effort ............................................. 1
2. Programming Model Overview ...................................... 1
3. Memory Model .................................................. 3
4. Execution Model ................................................ 4
   4.1 Progress of OpenSHMEM operations .......................... 4
   4.2 Atomicity Guarantees ....................................... 4
5. Language Bindings and Conformance ................................. 4
6. Library Constants .............................................. 5
7. Environment Variables ........................................... 6
8. OpenSHMEM Library API .......................................... 7
   8.1 Library Setup and Query Operations ....................... 7
       8.1.1 SHMEM ............................................... 7
       8.1.2 SHMEM ............................................... 7
       8.1.3 SHMEM ............................................... 8
       8.1.4 SHMEM ............................................... 8
       8.1.5 SHMEM ............................................... 9
       8.1.6 SHMEM ............................................... 10
       8.1.7 SHMEM ............................................... 11
   8.2 Memory Management Operations ............................... 13
       8.2.1 SHMALLOC, SHFREE, SHREALLOC, SHMEMALIGN ........ 13
       8.2.2 SHPALLOC ........................................ 14
       8.2.3 SHPCLMOVE ........................................ 15
       8.2.4 SHPDEALLC ......................................... 15
   8.3 Remote Memory Access Operations ............................. 16
       8.3.1 SHMEM_PUT ......................................... 17
       8.3.2 SHMEM_P ........................................... 19
       8.3.3 SHMEM_PUT ......................................... 20
       8.3.4 SHMEM_GET ......................................... 22
       8.3.5 SHMEM_G ........................................... 24
       8.3.6 SHMEM_IGET ......................................... 25
   8.4 Atomic Memory Operations .................................... 27
       8.4.1 SHMEM_ADD ......................................... 27
       8.4.2 SHMEM_CSWAP ...................................... 28
       8.4.3 SHMEM_SWAP ........................................ 30
       8.4.4 SHMEM_FINC ........................................ 31
       8.4.5 SHMEM_INC .......................................... 32
       8.4.6 SHMEM_FADD ........................................ 34
   8.5 Collective Operations ....................................... 35
       8.5.1 SHMEM_BARRIER_ALL .................................. 36
       8.5.2 SHMEM_BARRIER ...................................... 37
       8.5.3 SHMEM_BROADCAST .................................... 38
       8.5.4 SHMEM_COLLECT, SHMEM_FCOLLECT .................. 41
       8.5.5 SHMEM_REDUCTIONS ................................ 43
   8.6 Point-To-Point Synchronization Functions .................... 50
1. The OpenSHMEM Effort

OpenSHMEM is a Partitioned Global Address Space (PGAS) library interface specification. OpenSHMEM aims to provide a standard Application Programming Interface (API) for SHMEM libraries to aid portability and facilitate uniform predictable results of OpenSHMEM applications by explicitly stating the behavior and semantics of the OpenSHMEM library calls. Through the different versions, OpenSHMEM will continue to address the requirements of the PGAS community. As of this specification, existing vendors are moving towards OpenSHMEM compliant implementations and new vendors are developing OpenSHMEM library implementations to help the users write portable OpenSHMEM code. This ensures that applications can run on multiple platforms without having to deal with subtle vendor-specific implementation differences. For more details on the history of OpenSHMEM please refer to The History of OpenSHMEM section.

The OpenSHMEM effort is driven by the Extreme Scale Systems Center (ESSC) at ORNL and the University of Houston with significant input from the OpenSHMEM community. Besides the specification, the effort also includes providing a reference OpenSHMEM implementation, validation and verification suites, tools, a mailing list and website infrastructure to support specification activities. For more information please refer to: http://www.openshmem.org/.

2. Programming Model Overview

OpenSHMEM implements PGAS by defining remotely accessible data objects as mechanisms to share information among OpenSHMEM processes or Processing Elements (PEs) and private data objects that are accessible by the PE itself. The API allows communication and synchronization operations on both private (local) and remotely accessible data objects. The key feature of OpenSHMEM is that data transfer functions are one-sided in nature. This means that a local PE executing a data transfer does not require the participation of the remote PE to complete the operation. This allows for overlap between communication and computation to hide data transfer latencies, which makes OpenSHMEM ideal for unstructured, small/medium size data communication patterns. The OpenSHMEM library functions have the potential to provide low-latency, high-bandwidth communication API for use in highly parallelized scalable programs.

The OpenSHMEM interfaces can be used to implement Single Program Multiple Data (SPMD) style programs. It provides interfaces to start the OpenSHMEM PEs in parallel, and communication and synchronization interfaces to access remotely accessible data objects across PEs. These interfaces can be leveraged to divide a problem into multiple sub-problems that can solved independently or with coordination using the communication and synchronization interfaces. The OpenSHMEM specification defines library calls, constants, variables, and language bindings for C and Fortran. The C++ interface is currently the same as that for C. Unlike UPC, Fortran 2008, Titanium, X10 and Chapel, which are all PGAS languages, OpenSHMEM relies on the programmer to use the library calls to implement the correct semantics of its programming model.

An overview of the OpenSHMEM operations is described below:

1. Library Setup and Query

   (a) Initialization: The OpenSHMEM library environment is initialized.
   (b) Query: The local PE may get number of PEs running the same application and its unique integer identifier.
   (c) Accessibility: The local PE can find out if a remote PE is executing the same binary, or if a particular symmetric data object can be accessed by a remote PE, or may obtain a pointer to a symmetric data object on the specified remote PE on shared memory systems.

2. Symmetric Data Object Management

   1The OpenSHMEM specification is owned by Open Source Software Solutions Inc., a non-profit organization, under an agreement with SGI.
2. PROGRAMMING MODEL OVERVIEW

(a) **Allocation:** All executing PEs must participate in the allocation of a symmetric data object with identical arguments.

(b) **Deallocation:** All executing PEs must participate in the deallocation of the same symmetric data object with identical arguments.

(c) **Reallocation:** All executing PEs must participate in the reallocation of the same symmetric data object with identical arguments.

3. Remote Memory Access

(a) **Put:** The local PE specifies the source data (local or symmetric) that is copied to the symmetric data object on the remote PE.

(b) **Get:** The local PE specifies the symmetric data object on the remote PE that is copied to a data object (local or symmetric) on the local PE.

4. Atomics

(a) **Swap:** The PE initiating the swap gets the old value of the symmetric data object it is copying a new value to on the remote PE.

(b) **Increment:** The PE initiating the increment adds 1 to the symmetric data object on the remote PE.

(c) **Add:** The PE initiating the add specifies the value to be added to the symmetric data object on the remote PE.

(d) **Compare and Swap:** The PE initiating the swap gets the old value of the symmetric data object based on a value to be compared and copies a new value to the symmetric data object on the remote PE.

(e) **Fetch and Increment:** The PE initiating the increment adds 1 to the symmetric data object on the remote PE and returns with the old value.

(f) **Fetch and Add:** The PE initiating the add specifies the value to be added to the symmetric data object on the remote PE and returns with the old value.

5. Synchronization and Ordering

(a) **Fence:** The PE calling fence ensures ordering of remote access operations and stores to symmetric data objects with respect to a specific target PE.

(b) **Quiet:** The PE calling quiet ensures completion of remote access operations and stores to symmetric data objects.

(c) **Barrier:** All or some PEs collectively synchronize and ensure completion of all remote and local updates prior to any PE returning from the call.

6. Collective Communication

(a) **Broadcast:** The root PE specifies a symmetric data object to be copied to a symmetric data object on one or more remote PEs (not including itself).

(b) **Collection:** All PEs participating in the operation get the result of concatenated symmetric objects contributed by each of the PE in another symmetric data object.

(c) **Reduction:** All PEs participating in the operation get the result of associative binary operation over elements of the specified symmetric data object on another symmetric data object.

7. Mutual Exclusion

(a) **Set Lock:** The PE acquires exclusive access to the region bounded by the symmetric lock variable.

(b) **Test Lock:** The PE tests the symmetric lock variable for availability.

(c) **Clear Lock:** The PE which has previously acquired the lock releases it.

8. Data Cache Control (*deprecated on cache coherent systems*)

(a) Implementation of mechanisms to exploit the capabilities of hardware cache if available.
3 Memory Model

An OpenSHMEM program consists of data objects that are private to each PE and data objects that are remotely accessible by all PEs. Private data objects are stored in the local memory of each PE and can only be accessed by the PE itself; these data objects cannot be accessed by other PEs via OpenSHMEM routines. Private data objects follow the memory model of C or Fortran. Remotely accessible objects, however, can be accessed by remote PEs using OpenSHMEM routines. Remotely accessible data objects are called Symmetric Objects. All symmetric data objects have a corresponding object with the same name, type, size, and offset (from an arbitrary memory address) on all PEs. Symmetric objects are accessible by all executing PEs via the OpenSHMEM API. Symmetric data objects accessed via typed OpenSHMEM interfaces are required to be natural aligned based on their type requirements and underlying architecture. In OpenSHMEM the following kinds of data objects are symmetric:

- **Fortran** data objects in common blocks or with the SAVE attribute. These data objects must not be defined in a dynamic shared object (DSO).
- Global and static C and C++ variables. These data objects must not be defined in a DSO.
- **Fortran** arrays allocated with `shpalloc`
- C and C++ data allocated by `shmalloc`

OpenSHMEM dynamic memory allocation routines (`shpalloc` and `shmalloc`) allow collective allocation of Symmetric Data Objects on a special memory region called the Symmetric Heap. The Symmetric Heap is created during the execution of a program at a memory location determined by the implementation. The Symmetric Heap may reside on different memory regions on different PEs. Figure 1 shows how OpenSHMEM implements a PGAS model using remotely accessible (Symmetric objects) and private data objects when executing an OpenSHMEM program. Symmetric data objects are stored on the symmetric heap or in the global/static memory section of each PE.
4. Execution Model

An OpenSHMEM program consists of a set of OpenSHMEM processes called PEs that execute in a SPMD-like model where each PE can take a different execution path. A PE can be implemented using an OS process or an OS thread\(^2\). The PEs progress asynchronously, and can communicate/synchronize via the OpenSHMEM interfaces. All PEs in an OpenSHMEM program should start by calling the initialization function `start_pes` before using any of the other OpenSHMEM library routines. As of now, an OpenSHMEM program finishes execution by returning from the main function. On program exit, OpenSHMEM must complete all pending communication and release all the resources associated to the library using an implicit collective synchronization across PEs.

The PEs of the OpenSHMEM program are identified by unique integers. The identifiers are integers assigned in a monotonically increasing manner from zero to the total number of PEs minus 1. PE identifiers are used for OpenSHMEM calls (e.g. to specify `Put` or `Get` operations on symmetric data objects, collective synchronization calls, etc.) or to dictate a control flow for PEs using constructs of C or Fortran. The identifiers are fixed for the life cycle of the OpenSHMEM program.

4.1 Progress of OpenSHMEM operations

The OpenSHMEM model assumes that computation and communication are naturally overlapped. OpenSHMEM programs are expected to exhibit progression of communication both with and without OpenSHMEM calls. Consider a PE that is engaged in a computation with no OpenSHMEM calls. Other PEs should be able to communicate (`put`, `get`, `collective`, `atomic`, etc) and complete communication operations with that computationally-bound PE without that PE issuing any explicit OpenSHMEM calls. OpenSHMEM communication calls involving that PE should progress regardless of when that PE next engages in an OpenSHMEM call.

Note to implementors:

- An OpenSHMEM implementation for hardware that does not provide asynchronous communication capabilities may require a software progress thread in order to progress remotely-issued communication requests without explicit application calls to the OpenSHMEM library.
- High performance implementations of OpenSHMEM are expected to leverage hardware offload capabilities and provide asynchronous one-sided communication without software assistance.
- Implementations should avoid deferring the execution of one-sided operations until a synchronization point where data is known to be available. High-quality implementations should attempt asynchronous delivery whenever possible, for performance reasons. Additionally, the OpenSHMEM community discourages releasing OpenSHMEM implementations that do not provide asynchronous one-sided operations, since those have very limited performance value for OpenSHMEM applications.

4.2 Atomicity Guarantees

OpenSHMEM contains a number of routines that operate on symmetric data atomically (Section 8.4). These routines guarantee that accesses by OpenSHMEM’s atomic operations will be exclusive, but do not guarantee exclusivity in combination with other routines, either inside OpenSHMEM or outside.

For example: during the execution of an atomic remote integer increment operation on a symmetric variable \(X\), no other OpenSHMEM atomic operation may access \(X\). After the increment, \(X\) will have increased its value by \(l\) on the target PE, at which point other atomic operations may then modify that \(X\). However, access to the symmetric object \(X\) with non-atomic operations, such as one-sided `Put` or `Get` operations, will invalidate the atomicity guarantees.

5 Language Bindings and Conformance

OpenSHMEM provides ISO C and Fortran 90 language bindings. Any implementation that provides both C and

\(^2\)Implementing PEs using OS threads require compiler techniques to implement the OpenSHMEM memory model.
FürTRAN bindings can claim conformance to the specification. An implementation that provides e.g. only a C interface may claim to conform to the OpenSHMEM specification with respect to the C language, but not to FüRTRAN, and should make this clear in its documentation. The OpenSHMEM header files for C and FüRTRAN must contain only the interfaces and constant names defined in this specification.

OpenSHMEM APIs can be implemented as either functions or macros. However, implementing the interfaces using macros is strongly discouraged as this could severely limit the use of external profiling tools and high-level compiler optimizations. An OpenSHMEM program should avoid defining function names, variables, or identifiers with the prefix SHMEM_ (for C and FüRTRAN), _SHMEM_ (for C) or with OpenSHMEM API names.

All OpenSHMEM extension APIs that are not part of this specification must be defined in shmemx.h include file. These extensions shall use shmemx_ prefix for all function, variable, and constant names.

6 Library Constants

Constants Related To Collective Operations

Below are the library constants for collective operations. The constants that start with SHMEM_* are for FüRTRAN and _SHMEM_* for C.
### 7. Environment Variables

The OpenSHMEM specification provides a set of environment variables that allows users to configure the OpenSHMEM implementation, and receive information about the implementation. The implementations of the specification are free to define additional variables. Currently, the specification defines four environment variables.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMA_VERSION</td>
<td>any</td>
<td>print the library version at start-up</td>
</tr>
<tr>
<td>SMA_INFO</td>
<td>any</td>
<td>print helpful text about all these environment variables</td>
</tr>
<tr>
<td>SMA_SYMMETRIC_SIZE</td>
<td>non-negative integer</td>
<td>number of bytes to allocate for symmetric heap</td>
</tr>
<tr>
<td>SMA_DEBUG</td>
<td>any</td>
<td>enable debugging messages</td>
</tr>
</tbody>
</table>
8 OpenSHMEM Library API

8.1 Library Setup and Query Operations

The library setup and query interfaces that initialize and monitor the parallel environment of the PEs.

8.1.1 START_PES

Called at the beginning of an OpenSHMEM program to initialize the execution environment.

SYNOPSIS

C/C++:
```c
void start_pes(int npes);
```

FORTRAN:
```fortran
CALL START_PES(npes)
```

DESCRIPTION

Arguments

npes  Unused  Should be set to 0.

API description

The start_pes routine initializes the OpenSHMEM execution environment. An OpenSHMEM application must call start_pes before calling any other OpenSHMEM routine.

Return Values

None.

Notes

If any other OpenSHMEM call occurs before start_pes, the behavior is undefined. Although it is recommended to set npes to 0, this is not mandated. The value is ignored. Calling start_pes more than once has no subsequent effect.

EXAMPLES

This is a simple program that calls start_pes:

```plaintext
PROGRAM PUT

INTEGER TARG, SRC, RECEIVER, BAR
COMMON /T/ TARG
PARAMETER (RECEIVER=1)
CALL START_PES(0)

IF (MY_PE() .EQ. 0) THEN
  SRC = 33
  CALL SHMEM_INTEGER_PUT(TARG, SRC, 1, RECEIVER)
ENDIF

CALL SHMEM_BARRIER_ALL  ! SYNCHRONIZES SENDER AND RECEIVER

IF (MY_PE() .EQ. RECEIVER) THEN
  PRINT*, 'PE ', MY_PE(), ' TARG=',TARG,' (expect 33)'
ENDIF
END
```
8.1.2 SHMEM_MY_PE

Returns the number of the calling PE.

SYNOPSIS

C/C++:

```c
int shmem_my_pe(void);
int _my_pe (void);
```

FORTRAN:

```fortran
INTEGER SHMEM_MY_PE, ME
ME = SHMEM_MY_PE()
ME = MY_PE()
```

DESCRIPTION

Arguments

None

API description

This function returns the PE number of the calling PE. It accepts no arguments. The result is an integer between 0 and npes - 1, where npes is the total number of PEs executing the current program.

Return Values

Integer - Between 0 and npes - 1

Notes

Each PE has a unique number or identifier.

EXAMPLES

The following _my_pe example is for C/C++ programs:

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

int main(void)
{
    int me;

    start_pes(0);
    me = _my_pe();
    printf("My PE id is: %d\n", me);
    return 0;
}
```

8.1.3 SHMEM_N_PES

Returns the number of PEs running in an application.

SYNOPSIS

C/C++:
8. OPENSHMEM LIBRARY API

```c
int shmem_n_pes(void);
int _num_pes (void);
```

FORTRAN:
```
INTEGER SHMEM_N_PES, N_PES
N_PES = SHMEM_N_PES()
N_PES = NUM_PES()
```

DESCRIPTION

Arguments

None

API description

The function returns the number of PEs running the application.

Return Values

Integer - Number of PEs running the OpenSHMEM application.

Notes

None.

EXAMPLES

The following _num_pes example is for C/C++ programs:
```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    int npes;
    start_pes(0);
    npes = _num_pes();
    if (_my_pe() == 0) {
        printf("Number of PEs executing this application is: %d\n", npes);
    }
    return 0;
}
```

8.1.4 SHMEM_PE_ACCESSIBLE

Determines whether a PE is accessible via OpenSHMEM's data transfer operations.

SYNOPSIS

C/C++:
```
int shmem_pe_accessible(int pe);
```

FORTRAN:
```
LOGICAL LOG, SHMEM_PE_ACCESSIBLE
INTEGER pe
LOG = SHMEM_PE_ACCESSIBLE(pe)
```
8. OPENSHEM LIBRARY API

DESCRIPTION

Arguments

| IN     | pe     | Specific pe that needs to be checked if accessible from the local PE. |

API description

`shmem_pe_accessible` is a query function that indicates whether a specified PE is accessible via OpenSHMEM from the local PE. The `shmem_pe_accessible` function returns `TRUE` only if the remote PE is a process running from the same executable file as the local PE, indicating that full OpenSHMEM support for symmetric data objects (that resides in the static memory and symmetric heap) is available, otherwise it returns `FALSE`. This function may be particularly useful for hybrid programming with other communication libraries (such as a MPI) or parallel languages. For example, on SGI Altix series systems, OpenSHMEM is supported across multiple partitioned hosts and InfiniBand connected hosts. When running multiple executable MPI applications using OpenSHMEM on an Altix, full OpenSHMEM support is available between processes running from the same executable file. However, OpenSHMEM support between processes of different executable files is supported only for data objects on the symmetric heap, since static data objects are not symmetric between different executable files.

Return Values

C: The return value is 1 if the specified PE is a valid remote PE for OpenSHMEM functions; otherwise, it is 0.

Fortran: The return value is `TRUE` if the specified PE is a valid remote PE for OpenSHMEM functions; otherwise, it is `FALSE`.

Notes

None.

8.1.5 SHMEM_ADDR_ACCESSIBLE

Determines whether an address is accessible via OpenSHMEM data transfers operations from the specified remote PE.

SYNOPSIS

C/C++:

```c
int shmem_addr_accessible(void *addr, int pe);
```

FORTRAN:

```fortran
LOGICAL LOG, SHMEM_ADDR_ACCESSIBLE
INTEGER pe
LOG = SHMEM_ADDR_ACCESSIBLE(addr, pe)
```

DESCRIPTION

Arguments

| IN     | addr    | Data object on the local PE. |
| IN     | pe      | Integer id of a remote PE. |
API description

`shmem_addr_accessible` is a query function that indicates whether a local address is accessible via OpenSHMEM operations from the specified remote PE.

This function verifies that the data object is symmetric and accessible with respect to a remote PE via OpenSHMEM data transfer functions. The specified address `addr` is a data object on the local PE.

This function may be particularly useful for hybrid programming with other communication libraries (such as a MPI) or parallel languages. For example, in SGI Altix series systems, for multiple executable MPI applications that use OpenSHMEM functions, it is important to note that static memory, such as a Fortran common block or C global variable, is symmetric between processes running from the same executable file, but is not symmetric between processes running from different executable files. Data allocated from the symmetric heap (`shmalloc` or `shpalloc`) is symmetric across the same or different executable files.

Return Values

C/C++: The return value is 1 if `addr` is a symmetric data object and accessible via OpenSHMEM operations from the specified remote PE; otherwise, it is 0.

Fortran: The return value is .TRUE. if `addr` is a symmetric data object and accessible via OpenSHMEM operations from the specified remote PE; otherwise, it is .FALSE..

Notes

None.

8.1.6 SHMEM_PTR

Returns a pointer to a data object on a specified PE.

SYNOPSIS

C/C++:
```c
void *shmem_ptr(void *target, int pe);
```

FORTRAN:
```fortran
POINTER (PTR, POINTEE)
INTEGER pe
PTR = SHMEM_PTR(target, pe)
```

DESCRIPTION

Arguments

IN `target` The symmetric data object to be referenced.

IN `pe` An integer that indicates the PE number on which `target` is to be accessed. If you are using Fortran, it must be a default integer value.

API description

`shmem_ptr` returns an address that may be used to directly reference `target` on the specified PE. This address can be assigned to a pointer. After that, ordinary loads and stores to this remote address may be performed.

When a sequence of loads (gets) and stores (puts) to a data object on a remote PE does not match the access pattern provided in an OpenSHMEM data transfer routine like `shmem_put32` or `shmem_real_iget`, the `shmem_ptr` function can provide an efficient means to accomplish the communication.
Return Values

`shmemp_ptr` returns a pointer to the data object on the specified remote PE. If `target` is not remotely accessible, a `NULL` pointer is returned.

Notes

The `shmemp_ptr` function is available only on systems where ordinary memory loads and stores are used to implement OpenSHMEM put and get operations. When calling `shmemp_ptr`, you pass the address on the calling PE for a symmetric array on the remote PE.

EXAMPLES

This Fortran program calls `shmemp_ptr` and then PE 0 writes to the `BIGD` array on PE 1:

```fortran
PROGRAM REMOTEWRITE
INCLUDE 'shmem.fh'
INTEGER BIGD(100)
SAVE BIGD
INTEGER POINTEE(*)
POINTER (PTR,POINTEE)
CALL START_PES(0)

IF (MY_PE() .EQ. 0) THEN
    ! initialize PE 1’s BIGD array
    PTR = SHMEM_PTR(BIGD, 1) ! get address of PE 1’s BIGD
    ! array
    DO I=1,100
       POINTEE(I) = I
    ENDDO
ENDIF
CALL SHMEM_BARRIER_ALL
IF (MY_PE() .EQ. 1) THEN
    PRINT*, 'BIGD on PE 1 is: '
    PRINT*, BIGD
ENDIF
END
```

This is the equivalent program written in C:

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

int main(void)
{
    static int bigd[100];
    int *ptr;
    int i;
    start_pes(0);
    if (_my_pe() == 0) {
        /* initialize PE 1’s bigd array */
        ptr = shmem_ptr(bigd, 1);
        if (ptr == NULL)
            printf("can’t use pointer to directly access PE 1’s array\n");
        else
            for (i=0; i<100; i++)
                *ptr++ = i+1;
    }
```
8. OPENSHMEM LIBRARY API

```c
shmem_barrier_all();

if (_my_pe() == 1) {
    printf("bigr on PE 1 is:\n");
    for (i=0; i<100; i++)
        printf("%d\n", bigd[i]);
    printf("\n");
}
return 1;
```

8.2 Memory Management Operations

OpenSHMEM provides a set of APIs for managing the symmetric heap. The APIs allow to dynamically allocate, deallocate, reallocate and align symmetric data objects in the symmetric heap, in C and Fortran.

8.2.1 SHMALLOC, SHFREE, SHREALLOC, SHMEMALIGN

Symmetric heap memory management functions.

**SYNOPSIS**

C/C++:

```c
void *shmalloc(size_t size);
void shfree(void *ptr);
void *shrealloc(void *ptr, size_t size);
void *shmemalign(size_t alignment, size_t size);
```

**DESCRIPTION**

**Arguments**

<table>
<thead>
<tr>
<th>IN</th>
<th>size</th>
<th>In bytes, to request a block to be allocated from the symmetric heap.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>ptr</td>
<td>Points to a block within the symmetric heap.</td>
</tr>
<tr>
<td>IN</td>
<td>alignment</td>
<td>Byte alignment of the block allocated from the symmetric heap.</td>
</tr>
</tbody>
</table>

**API description**

The *shmalloc* function returns a pointer to a block of at least size bytes suitably aligned for any use. This space is allocated from the symmetric heap (in contrast to *malloc*, which allocates from the private heap).

The *shmemalign* function allocates a block in the symmetric heap that has a byte alignment specified by the alignment argument.

The *shfree* function causes the block to which *ptr* points to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs.

The *shrealloc* function changes the size of the block to which *ptr* points to the size (in bytes) specified by *size*. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the block is indeterminate. If *ptr* is a NULL pointer, the *shrealloc* function behaves like the *shmalloc* function for the specified size. If *size* is 0 and *ptr* is not a NULL pointer, the block to which it points is freed. If the space cannot be allocated, the block to which *ptr* points is unchanged.

The *shmalloc*, *shfree*, and *shrealloc* functions are provided so that multiple PEs in an application can allocate symmetric, remotely accessible memory blocks. These memory blocks can then be used with OpenSHMEM communication routines. Each of these functions call the *shmem_barrier_all* function before returning; this ensures that all PEs participate in the memory allocation, and that the memory on other PEs can be used as soon as the local PE returns. The user is responsible for calling these functions with identical
argument(s) on all PEs; if differing size arguments are used, the behavior of the call and any subsequent OpenSHMEM calls becomes undefined.

Return Values

The *shmalloc* function returns a pointer to the allocated space (which should be identical on all PEs); otherwise, it returns a NULL pointer.

The *shfree* function returns no value.

The *shrealloc* function returns a pointer to the allocated space (which may have moved); otherwise, it returns a null pointer.

Notes

The total size of the symmetric heap is determined at job startup. One can adjust the size of the heap using the `SMA_SYMMETRIC_SIZE` environment variable (where available).

The *shmalloc*, *shfree*, and *shrealloc* functions differ from the private heap allocation functions in that all PEs in an application must call them (a barrier is used to ensure this).

8.2.2 SHPALLOC

Allocates a block of memory from the symmetric heap.

SYNOPSIS

FORTRAN:

```fortran
POINTER (addr, A(1))
INTEGER length, errcode, abort
CALL SHPALLOC(addr, length, errcode, abort)
```

DESCRIPTION

Arguments

| OUT  | addr | First word address of the allocated block. |
| IN   | length | Number of words of memory requested. One word is 32 bits. |
| OUT  | errcode | Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error. |
| IN   | abort | Abort code; nonzero requests abort on error; 0 requests an error code. |

API description

*SHPALLOC* allocates a block of memory from the program’s symmetric heap that is greater than or equal to the size requested. To maintain symmetric heap consistency, all PEs in an program must call *SHPALLOC* with the same value of length; if any PEs are missing, the program will hang.

By using the Fortran POINTER mechanism in the following manner, you can use array *A* to refer to the block allocated by *SHPALLOC: POINTER (addr, A())*

Return Values

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Length is not an integer greater than 0</td>
</tr>
<tr>
<td>-2</td>
<td>No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).</td>
</tr>
</tbody>
</table>
Notes
The total size of the symmetric heap is determined at job startup. One may adjust the size of the heap using the \texttt{SMA\_SYMMETRIC\_SIZE} environment variable (if available).

8.2.3 \texttt{SHPCLMOVE}

Extends a symmetric heap block or copies the contents of the block into a larger block.

SYNOPSIS

FORTRAN:

\begin{verbatim}
POINTER (addr, A(1))
INTEGER length, status, abort
CALL SHPCLMOVE (addr, length, status, abort)
\end{verbatim}

DESCRIPTION

Arguments

\begin{description}
\item [INOUT] \texttt{addr} & On entry, first word address of the block to change; on exit, the new address of the block if it was moved.
\item [IN] \texttt{length} & Requested new total length in words. One word is 32 bits.
\item [OUT] \texttt{status} & Status is 0 if the block was extended in place, 1 if it was moved, and a negative integer for the type of error detected.
\item [IN] \texttt{abort} & Abort code. Nonzero requests abort on error; 0 requests an error code.
\end{description}

API description

The \texttt{SHPCLMOVE} function either extends a symmetric heap block if the block is followed by a large enough free block or copies the contents of the existing block to a larger block and returns a status code indicating that the block was moved. This function also can reduce the size of a block if the new length is less than the old length. All PEs in a program must call \texttt{SHPCLMOVE} with the same value of \texttt{addr} to maintain symmetric heap consistency; if any PEs are missing, the program hangs.

Return Values

\begin{tabular}{|c|c|}
\hline
Error Code & Condition \tabularnewline
\hline
-1 & Length is not an integer greater than 0 \tabularnewline
-2 & No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap). \tabularnewline
-3 & Address is outside the bounds of the symmetric heap. \tabularnewline
-4 & Block is already free. \tabularnewline
-5 & Address is not at the beginning of a block. \tabularnewline
\hline
\end{tabular}

Notes
None.

8.2.4 \texttt{SHPDEALLC}

Returns a memory block to the symmetric heap.
SYNOPSIS

FORTRAN:

POINTER (addr, A(1))
INTEGER errcode, abort
CALL SHPDEALLC(addr, errcode, abort)

DESCRIPTION

Arguments

IN    addr   First word address of the block to deallocate.
OUT   errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error.
IN    abort  Abort code. Nonzero requests abort on error; 0 requests an error code.

API description

SHPDEALLC returns a block of memory (allocated using SHPALLOC) to the list of available space in the symmetric heap. To maintain symmetric heap consistency, all PEs in a program must call SHPDEALLC with the same value of addr; if any PEs are missing, the program hangs.

Return Values

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Length is not an integer greater than 0</td>
</tr>
<tr>
<td>-2</td>
<td>No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).</td>
</tr>
<tr>
<td>-3</td>
<td>Address is outside the bounds of the symmetric heap.</td>
</tr>
<tr>
<td>-4</td>
<td>Block is already free.</td>
</tr>
<tr>
<td>-5</td>
<td>Address is not at the beginning of a block.</td>
</tr>
</tbody>
</table>

Notes

None.

8.3 Remote Memory Access Operations

Remote Memory Access (RMA) operations described in this section are one-sided communication mechanisms of the OpenSHMEM API. While using these mechanisms, the programmer is required to provide parameters only on the calling side. A characteristic of one-sided communication is that it decouples communication from the synchronization. One-sided communication mechanisms transfer the data but do not synchronize the sender of the data with the receiver of the data.

OpenSHMEM RMA operations are all performed on the symmetric objects. The initiator PE of the call is designated as source, and the PE in which memory is accessed is designated as target. In the case of the remote update operation, Put, the origin is the source PE and the destination PE is the target PE. In the case of the remote read operation, Get, the origin is the target PE and the destination is the source PE.

OpenSHMEM provides three different types of one-sided communication interfaces. shmem_put<bits> interface transfers data in chunks of bits. shmem_put32, for example, copies data to a target PE in chunks of 32 bits. shmem_<datatype>_put interface copies elements of type datatype from a source PE to a target PE. For example, shmem_integer_put, copies elements of type integer from a source PE to a target PE. shmem_<datatype>_p interface is similar to shmem_<datatype>_put except that it only transfers one element of type datatype.
OpenSHMEM provides interfaces for transferring both contiguous and non-contiguous data. The non-contiguous data transfer interfaces are prefixed with “i”. *shmem_*<datatype>*_iput* interface, for example, copies strided data elements from the *source* PE to a *target* PE.

### 8.3.1 SHMEM_PUT

The put routines provide a method for copying data from a contiguous local data object to a data object on a specified PE.

**SYNOPSIS**

**C/C++:**
```c
void shmem_double_put(double *target, const double *source, size_t len, int pe);
void shmem_float_put(float *target, const float *source, size_t len, int pe);
void shmem_int_put(int *target, const int *source, size_t len, int pe);
void shmem_long_put(long *target, const long *source, size_t len, int pe);
void shmem_longdouble_put(long double *target, const long double *source, size_t len, int pe);
void shmem_longlong_put(long long *target, const long long *source, size_t len, int pe);
void shmem_put32(void *target, const void *source, size_t len, int pe);
void shmem_put64(void *target, const void *source, size_t len, int pe);
void shmem_put128(void *target, const void *source, size_t len, int pe);
void shmem_putmem(void *target, const void *source, size_t len, int pe);
void shmem_short_put(short *target, const short *source, size_t len, int pe);
```

**FORTRAN:**
```fortran
CALL SHMEM_CHARACTER_PUT(target, source, len, pe)
CALL SHMEM_COMPLEX_PUT(target, source, len, pe)
CALL SHMEM_DOUBLE_PUT(target, source, len, pe)
CALL SHMEM_INTEGER_PUT(target, source, len, pe)
CALL SHMEM_LOGICAL_PUT(target, source, len, pe)
CALL SHMEM_PUT(target, source, len, pe)
CALL SHMEM_PUT4(target, source, len, pe)
CALL SHMEM_PUT8(target, source, len, pe)
CALL SHMEM_PUT32(target, source, len, pe)
CALL SHMEM_PUT64(target, source, len, pe)
CALL SHMEM_PUT128(target, source, len, pe)
CALL SHMEM_PUTMEM(target, source, len, pe)
CALL SHMEM_REAL_PUT(target, source, len, pe)
```

**DESCRIPTION**

**Arguments**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>target</td>
<td>Data object to be updated on the remote PE. This data object must be remotely accessible.</td>
</tr>
<tr>
<td>OUT</td>
<td>source</td>
<td>Data object containing the data to be copied.</td>
</tr>
<tr>
<td>IN</td>
<td>len</td>
<td>Number of elements in the <em>target</em> and <em>source</em> arrays. <em>len</em> must be of type <code>size_t</code> for C. If you are using Fortran, it must be a constant, variable, or array element of default integer type.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>PE number of the remote PE. <em>pe</em> must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.</td>
</tr>
</tbody>
</table>

**API description**

The routines return after the data has been copied out of the *source* array on the local PE. The delivery of
data words into the data object on the destination PE may occur in any order. Furthermore, two successive put operations may deliver data out of order unless a call to \textit{shmem\_fence} is introduced between the two calls.

The \textit{target} and \textit{source} data objects must conform to certain typing constraints, which are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{shmem_putmem}</td>
<td>\textit{Fortran}: Any noncharacter type. \textit{C}: Any data type. \textit{len} is scaled in bytes.</td>
</tr>
<tr>
<td>\textit{shmem_put4}, \textit{shmem_put32}</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
<tr>
<td>\textit{shmem_put8}, \textit{shmem_put64}</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>\textit{shmem_put128}</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>\textit{shmem_double_put}</td>
<td>Elements of type double.</td>
</tr>
<tr>
<td>\textit{shmem_longdouble_put}</td>
<td>Elements of type long double.</td>
</tr>
<tr>
<td>\textit{SHMEM_CHARACTER_PUT}</td>
<td>Elements of type character. \textit{len} is the number of characters to transfer. The actual character lengths of the \textit{source} and \textit{target} variables are ignored.</td>
</tr>
<tr>
<td>\textit{SHMEM_COMPLEX_PUT}</td>
<td>Elements of type complex of default size.</td>
</tr>
<tr>
<td>\textit{SHMEM_DOUBLE_PUT}</td>
<td>Elements of type double precision.</td>
</tr>
<tr>
<td>\textit{SHMEM_INTEGER_PUT}</td>
<td>Elements of type integer.</td>
</tr>
<tr>
<td>\textit{SHMEM_LOGICAL_PUT}</td>
<td>Elements of type logical.</td>
</tr>
<tr>
<td>\textit{SHMEM_REAL_PUT}</td>
<td>Elements of type real.</td>
</tr>
</tbody>
</table>

\textbf{Return Values}

None.

\textbf{Notes}

If you are using \textit{Fortran}, data types must be of default size. For example, a real variable must be declared as \textit{REAL}, \textit{REAL*4}, or \textit{REAL(KIND=KIND(1.0))}.

\textbf{EXAMPLES}

The following \textit{shmem\_put} example is for programs:

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

int main(void)
{
    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 */
    printf("target[0] on PE %d is %d\n", _my_pe(), target[0]);
    return 1;
}
```
8. OPENSHMEM LIBRARY API

8.3.2 SHMEM_P

Copies one data item to a remote PE.

SYNOPSIS

C/C++:

void shmem_char_p(char *addr, char value, int pe);
void shmem_short_p(short *addr, short value, int pe);
void shmem_int_p(int *addr, int value, int pe);
void shmem_long_p(long *addr, long value, int pe);
void shmem_longlong_p(long long *addr, long long value, int pe);
void shmem_float_p(float *addr, float value, int pe);
void shmem_double_p(double *addr, double value, int pe);
void shmem_longdouble_p(long double *addr, long double value, int pe);

DESCRIPTION

Arguments

IN addr The remotely accessible array element or scalar data object which will receive the data on the remote PE.
IN value The value to be transferred to addr on the remote PE.
IN pe The number of the remote PE.

API description

These routines provide a very low latency put capability for single elements of most basic types. As with shmem_put, these functions start the remote transfer and may return before the data is delivered to the remote PE. Use shmem_quiet to force completion of all remote Put transfers.

Return Values

None.

Notes

None.

EXAMPLES

The following example uses shmem_double_p in a C program.

#include <stdio.h>
#include <math.h>
#include <shmem.h>
static const double e = 2.71828182;
static const double epsilon = 0.00000001;

int main(void)
{
    double *f;
    int me;

    start_pes(0);
    me = _my_pe();
    f = (double *) shmalloc(sizeof (*f));
    *f = 3.1415927;
    shmem_barrier_all();
8. OPENSHMEM LIBRARY API

if (me == 0)
    shmem_double_p(f, e, 1);
shmem_barrier_all();
if (me == 1)
    printf("%s\n", (fabs(*f - e) < epsilon) ? "OK" : "FAIL");
return 0;
}

8.3.3 SHMEM_IPUT

Copies strided data to a specified PE.

SYNOPSIS

C/C++:

void shmem_double_iput(double *target, const double *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_float_iput(float *target, const float *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_int_iput(int *target, const int *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_iput32(void *target, const void *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_iput64(void *target, const void *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_iput128(void *target, const void *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_long_iput(long *target, const long *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_longdouble_iput(long double *target, const long double *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_longlong_iput(long long *target, const long long *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_short_iput(short *target, const short *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);

FORTRAN:

INTEGER tst, sst, nelems, pe
CALL SHMEM_COMPLEX_IPUT(target, source, tst, sst, nelems, pe)
CALL SHMEM_DOUBLE_IPUT(target, source, tst, sst, nelems, pe)
CALL SHMEM_INTEGER_IPUT(target, source, tst, sst, nelems, pe)
CALL SHMEM_IPUT4(target, source, tst, sst, nelems, pe)
CALL SHMEM_IPUT8(target, source, tst, sst, nelems, pe)
CALL SHMEM_IPUT32(target, source, tst, sst, nelems, pe)
CALL SHMEM_IPUT64(target, source, tst, sst, nelems, pe)
CALL SHMEM_IPUT128(target, source, tst, sst, nelems, pe)
CALL SHMEM_LOGICAL_IPUT(target, source, tst, sst, nelems, pe)
CALL SHMEM_REAL_IPUT(target, source, tst, sst, nelems, pe)

DESCRIPTION

Arguments

OUT target Array to be updated on the remote PE. This data object must be remotely accessible.
8. OPENSHMEM LIBRARY API

IN  source  Array containing the data to be copied.
IN  tst      The stride between consecutive elements of the target array. The stride is scaled by the element size of the target array. A value of 1 indicates contiguous data. tst must be of type ptrdiff_t. If you are using Fortran, it must be a default integer value.
IN  sst      The stride between consecutive elements of the source array. The stride is scaled by the element size of the source array. A value of 1 indicates contiguous data. sst must be of type ptrdiff_t. If you are using Fortran, it must be a default integer value.
IN  nelems   Number of elements in the target and source arrays. nelems must be of type size_t for C. If you are using Fortran, it must be a constant, variable, or array element of default integer type.
IN  pe       PE number of the remote PE. pe must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.

**API description**

The iput routines provide a method for copying strided data elements (specified by sst) of an array from a source array on the local PE to locations specified by stride tst on a target array on specified remote PE. Both strides, tst and sst must be greater than or equal to 1. The routines return when the data has been copied out of the source array on the local PE but not necessarily before the data has been delivered to the remote data object.

The target and source data objects must conform to typing constraints, which are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmep_iput32, shmep_iput4</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
<tr>
<td>shmep_iput64, shmep_iput8</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmep_iput128</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>shmep_short_iput</td>
<td>Elements of type short.</td>
</tr>
<tr>
<td>shmep_int_iput</td>
<td>Elements of type short.</td>
</tr>
<tr>
<td>shmep_long_iput</td>
<td>Elements of type long.</td>
</tr>
<tr>
<td>shmep_longlong_iput</td>
<td>Elements of type long long.</td>
</tr>
<tr>
<td>shmep_float_iput</td>
<td>Elements of type float.</td>
</tr>
<tr>
<td>shmep_double_iput</td>
<td>Elements of type double.</td>
</tr>
<tr>
<td>shmep_longdouble_iput</td>
<td>Elements of type long double.</td>
</tr>
<tr>
<td>SHMEM_COMPLEX_IPUT</td>
<td>Elements of type complex of default size.</td>
</tr>
<tr>
<td>SHMEM_DOUBLE_IPUT</td>
<td>Elements of type double precision.</td>
</tr>
<tr>
<td>SHMEM_INTEGER_IPUT</td>
<td>Elements of type integer.</td>
</tr>
<tr>
<td>SHMEM_LOGICAL_IPUT</td>
<td>Elements of type logical.</td>
</tr>
<tr>
<td>SHMEM_REAL_IPUT</td>
<td>Elements of type real.</td>
</tr>
</tbody>
</table>

**Return Values**

None.

**Notes**

If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4 or REAL(KIND=KIND(1.0)). See Introduction for a definition of the term remotely accessible.

**EXAMPLES**
Consider the following `shmem_long_iput` example for C/C++ programs.

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

int main(void)
{
    short source[10] = { 1, 2, 3, 4, 5,
                        6, 7, 8, 9, 10 };
    static short target[10];
    start_pes(0);
    if (_my_pe() == 0) {
        /* put 10 words into target on PE 1 */
        shmem_short_iput(target, source, 1, 2, 5, 1);
    }
    shmem_barrier_all(); /* sync sender and receiver */
    if (_my_pe() == 1) {
        printf("target on PE %d is %d %d %d %d %d
", _my_pe(),
                (int)target[0], (int)target[1], (int)target[2],
                (int)target[3], (int)target[4]);
    }
    shmem_barrier_all(); /* sync before exiting */
    return 1;
}
```

### 8.3.4 SHMEM\_GET

Copies data from a specified PE.

**SYNOPSIS**

**C/C++:**

```c
void shmem_double_get(double *target, const double *source, size_t nelems, int pe);
void shmem_float_get(float *target, const float *source, size_t nelems, int pe);
void shmem_int32_get(void *target, const void *source, size_t nelems, int pe);
void shmem_int64_get(void *target, const void *source, size_t nelems, int pe);
void shmem_getmem(void *target, const void *source, size_t nelems, int pe);
void shmem_int_get(int *target, const int *source, size_t nelems, int pe);
void shmem_long_get(long *target, const long *source, size_t nelems, int pe);
void shmem_longdouble_get(long double *target, const long double *source, size_t nelems, int pe);
void shmem_longlong_get(long long *target, const long long *source, size_t nelems, int pe);
void shmem_short_get(short *target, const short *source, size_t nelems, int pe);
```

**FORTRAN:**

```fortran
INTEGER nelems, pe
CALL SHMEM\_CHARACTER\_GET(target, source, nelems, pe)
CALL SHMEM\_COMPLEX\_GET(target, source, nelems, pe)
CALL SHMEM\_DOUBLE\_GET(target, source, nelems, pe)
CALL SHMEM\_GET4(target, source, nelems, pe)
CALL SHMEM\_GET8(target, source, nelems, pe)
CALL SHMEM\_GET12(target, source, nelems, pe)
CALL SHMEM\_GET128(target, source, nelems, pe)
CALL SHMEM\_INTEGER\_GET(target, source, nelems, pe)
CALL SHMEM\_LOGICAL\_GET(target, source, nelems, pe)
CALL SHMEM\_REAL\_GET(target, source, nelems, pe)
```
DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>OUT</th>
<th>target</th>
<th>Local data object to be updated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>source</td>
<td>Data object on the PE identified by pe that contains the data to be copied. This data object must be remotely accessible.</td>
</tr>
<tr>
<td>IN</td>
<td>nelems</td>
<td>Number of elements in the target and source arrays. nelems must be of type size_t for C. If you are using Fortran, it must be a constant, variable, or array element of default integer type.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>PE number of the remote PE. pe must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.</td>
</tr>
</tbody>
</table>

API description

The get routines provide a method for copying a contiguous symmetric data object from a different PE to a contiguous data object on a the local PE. The routines return after the data has been delivered to the target array on the local PE.

The target and source data objects must conform to typing constraints, which are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_getmem</td>
<td>Fortran: Any noncharacter type. C: Any data type. nelems is scaled in bytes.</td>
</tr>
<tr>
<td>shmem_get4, shmem_get32</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
<tr>
<td>shmem_get8, shmem_get64</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_get128</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>shmem_short_get</td>
<td>Elements of type short.</td>
</tr>
<tr>
<td>shmem_int_get</td>
<td>Elements of type int.</td>
</tr>
<tr>
<td>shmem_long_get</td>
<td>Elements of type long.</td>
</tr>
<tr>
<td>shmem_longlong_get</td>
<td>Elements of type long long.</td>
</tr>
<tr>
<td>shmem_float_get</td>
<td>Elements of type float.</td>
</tr>
<tr>
<td>shmem_double_get</td>
<td>Elements of type double.</td>
</tr>
<tr>
<td>shmem_longdouble_get</td>
<td>Elements of type long double.</td>
</tr>
<tr>
<td>SHMEM_CHARACTER_GET</td>
<td>Elements of type character. nelems is the number of characters to transfer. The actual character nelemsgths of the source and target variables are ignored.</td>
</tr>
<tr>
<td>SHMEM_COMPLEX_GET</td>
<td>Elements of type complex of default size.</td>
</tr>
<tr>
<td>SHMEM_DOUBLE_GET</td>
<td>Fortran: Elements of type double precision.</td>
</tr>
<tr>
<td>SHMEM_INTEGER_GET</td>
<td>Elements of type integer.</td>
</tr>
<tr>
<td>SHMEM_LOGICAL_GET</td>
<td>Elements of type logical.</td>
</tr>
<tr>
<td>SHMEM_REAL_GET</td>
<td>Elements of type real.</td>
</tr>
</tbody>
</table>

Return Values

None.

Notes

See Introduction for a definition of the term remotely accessible. If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4, or REAL(KIND=KIND(1.0)).

EXAMPLES

Consider this example for Fortran.
8. OPENSHMEM LIBRARY API

8.3.5 SHMEM_G

Transfers one data item from a remote PE

SYNOPSIS

C/C++:

char shmem_char_g(char *addr, int pe);
short shmem_short_g(short *addr, int pe);
int shmem_int_g(int *addr, int pe);
long shmem_long_g(long *addr, int pe);
long long shmem_longlong_g(long long *addr, int pe);
float shmem_float_g(float *addr, int pe);
double shmem_double_g(double *addr, int pe);
long double shmem_longdouble_g(long double *addr, int pe);

DESCRIPTION

Arguments

IN | addr | The remotely accessible array element or scalar data object.
IN | pe | The number of the remote PE on which addr resides.

API description

These routines provide a very low latency get capability for single elements of most basic types.

Return Values

Returns a single element of type specified in the synopsis.

Notes

None.

EXAMPLES

The following shmem_long_g example is for C/C++ programs:

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

long x = 10101;
```
int main(void)
{
    int me, npes;
    long y = -1;

    start_pes(0);
    me = _my_pe();
    npes = _num_pes();

    if (me == 0)
        y = shmem_long_g(&x, npes-1);

    printf("%d: y = %ld\n", me, y);

    return 0;
}

8.3.6 SHMEM_IGET

Copies strided data from a specified PE.

SYNOPSIS

C/C++:

```c
void shmem_double_iget(double *target, const double *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_float_iget(float *target, const float *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_iget32(void *target, const void *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_iget64(void *target, const void *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_iget128(void *target, const void *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_int_iget(int *target, const int *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_long_iget(long *target, const long *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_longdouble_iget(long double *target, const long double *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_longlong_iget(long long *target, const long long *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
void shmem_short_iget(short *target, const short *source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe);
```

FORTRAN:

```fortran
INTEGER tst, sst, nelems, pe
CALL SHMEM_COMPLEX_IGET(target, source, tst, sst, nelems, pe)
CALL SHMEM_DOUBLE_IGET(target, source, tst, sst, nelems, pe)
CALL SHMEM_INTEGER_IGET(target, source, tst, sst, nelems, pe)
CALL SHMEM_LOGICAL_IGET(target, source, tst, sst, nelems, pe)
CALL SHMEM_REAL_IGET(target, source, tst, sst, nelems, pe)
CALL SHMEM_REAL128_IGET(target, source, tst, sst, nelems, pe)
```
DESCRIPTION

Arguments

| OUT | target     | Array to be updated on the local PE.                  |
| IN  | source     | Array containing the data to be copied on the remote PE. |
| IN  | tst        | The stride between consecutive elements of the target array. The stride is scaled by the element size of the target array. A value of 1 indicates contiguous data. tst must be of type ptrdiff_t. If you are calling from Fortran, it must be a default integer value. |
| IN  | sst        | The stride between consecutive elements of the source array. The stride is scaled by the element size of the source array. A value of 1 indicates contiguous data. sst must be of type ptrdiff_t. If you are calling from Fortran, it must be a default integer value. |
| IN  | nelems     | Number of elements in the target and source arrays. nelems must be of type size_t for C. If you are using Fortran, it must be a constant, variable, or array element of default integer type. |
| IN  | pe         | PE number of the remote PE. pe must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type. |

API description

The iget routines provide a method for copying strided data elements from a symmetric array from a specified remote PE to strided locations on a local array. The routines return when the data has been copied into the local target array.

The target and source data objects must conform to typing constraints, which are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_iget32, shmem_iget4</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
<tr>
<td>shmem_iget64, shmem_iget8</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_iget128</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>shmem_short_iget</td>
<td>Elements of type short.</td>
</tr>
<tr>
<td>shmem_int_iget</td>
<td>Elements of type int.</td>
</tr>
<tr>
<td>shmem_long_iget</td>
<td>Elements of type long.</td>
</tr>
<tr>
<td>shmem_longlong_iget</td>
<td>Elements of type long long.</td>
</tr>
<tr>
<td>shmem_float_iget</td>
<td>Elements of type float.</td>
</tr>
<tr>
<td>shmem_double_iget</td>
<td>Elements of type double.</td>
</tr>
<tr>
<td>shmem_longdouble_iget</td>
<td>Elements of type long double.</td>
</tr>
<tr>
<td>SHMEM_COMPLEX_IGET</td>
<td>Elements of type complex of default size.</td>
</tr>
<tr>
<td>SHMEM_DOUBLE_IGET</td>
<td>Fortran: Elements of type double precision.</td>
</tr>
<tr>
<td>SHMEM_INTEGER_IGET</td>
<td>Elements of type integer.</td>
</tr>
<tr>
<td>SHMEM_LOGICAL_IGET</td>
<td>Elements of type logical.</td>
</tr>
<tr>
<td>SHMEM_REAL_IGET</td>
<td>Elements of type real.</td>
</tr>
</tbody>
</table>

Return Values

None.

Notes

If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4, or REAL(KIND=KIND(1.0)).

EXAMPLES
The following example uses `shmem_logical_iget` in a Fortran program.

```fortran
PROGRAM STRIDELOGICAL
LOGICAL SOURCE(10), TARGET(5)
SAVE SOURCE ! SAVE MAKES IT REMOTELY ACCESSIBLE
DATA TARGET / 5*.F. /
CALL START_PES(2)
IF (MY_PE() .EQ. 0) THEN
   CALL SHMEM_LOGICAL_IGET(TARGET, SOURCE, 1, 2, 5, 1)
   PRINT*, 'TARGET AFTER SHMEM_LOGICAL_IGET:', TARGET
ENDIF
CALL SHMEM_BARRIER_ALL
```

### 8.4 Atomic Memory Operations

**Atomic Memory Operation** (AMO) is a one-sided communication mechanism that combines memory update operations with atomicity guarantees described in Section 4.2. Similar to the RMA routines, described in Section 8.3, the AMOs are performed only on symmetric objects. OpenSHMEM defines the two types of AMO routines:

- **The fetch-and-operate routines** combine memory update and fetch operations in a single atomic operation. The routines return after the data has been fetched and delivered to the local PE.
  
  The fetch-and-operate operations include: `SHMEM_CSWAP`, `SHMEM_SWAP`, `SHMEM_FINC`, and `SHMEM_FADD`.

- **The non-fetch atomic routines** update the remote memory in a single atomic operation. A non-fetch atomic routine starts the atomic operation and may return before the operation execution on the remote PE. To force completion for these non-fetch atomic routines, `shmem_quiet`, `shmem_barrier`, or `shmem_barrierall` can be used by an OpenSHMEM program.

  The non-fetch operations include: `SHMEM_INC` and `SHMEM_ADD`.

#### 8.4.1 SHMEM_ADD

Performs an atomic add operation on a remote symmetric data object.

**SYNOPSIS**

**C/C++:**

```c
void shmem_int_add(int *target, int value, int pe);
void shmem_long_add(long *target, long value, int pe);
void shmem_longlong_add(long long *target, long long value, int pe);
```

**FORTRAN:**

```fortran
INTEGER pe
INTEGER*4 value_i4
CALL SHMEM_INT4_ADD(target, value_i4, pe)
INTEGER*8 value_i8
CALL SHMEM_INT8_ADD(target, value_i8, pe)
```

**DESCRIPTION**

**Arguments**

| OUT | target | The remotely accessible integer data object to be updated on the remote PE. If you are using C/C++, the type of target should match that implied in the SYNOPSIS section. |
IN value

The value to be atomically added to target. If you are using C/C++, the type of value should match that implied in the SYNOPSIS section. If you are using Fortran, it must be of type integer with an element size of target.

IN pe

An integer that indicates the PE number upon which target is to be updated. If you are using Fortran, it must be a default integer value.

API description

The shmem_add routine performs an atomic add operation. It adds value to target on PE pe and atomically increments the target without returning the value.

If you are using Fortran, target must be of the following type:

+ **Routine** | **Data Type of target and source**
+-------------|---------------------------------|
+ SHMEM_INT4_ADD | 4-byte integer
+ SHMEM_INT8_ADD | 8-byte integer

Return Values

None.

Notes

The term remotely accessible is defined in the Introduction.

EXAMPLES

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

int main(void)
{
    int me, old;
    static int dst;
    start_pes(0);
    me = _my_pe();
    old = -1;
    dst = 22;
    shmem_barrier_all();
    if (me == 1){
        old = shmem_int_fadd(&dst, 44, 0);
    }
    shmem_barrier_all();
    printf("%d: old = %d, dst = %d
", me, old, dst);
    return 0;
}
```

8.4.2 SHMEM_CSWAP

Performs an atomic conditional swap to a remote data object.

SYNOPSIS

C/C++:
8. OPENSHMEM LIBRARY API

int shmem_int_cswap(int *target, int cond, int value, int pe);
long shmem_long_cswap(long *target, long cond, long value, int pe);
long shmem_longlong_cswap(long long *target, long long cond, long long value, int pe);

FORTRAN:

INTEGER pe
INTEGER*4 SHMEM_INT4_CSWAP, cond_i4, value_i4, ires_i4
ires_i4 = SHMEM_INT4_CSWAP(target, cond_i4, value_i4, pe)
INTEGER*8 SHMEM_INT8_CSWAP, cond_i8, value_i8, ires_i8
ires_i8 = SHMEM_INT8_CSWAP(target, cond_i8, value_i8, pe)

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>OUT</th>
<th>target</th>
<th>The remotely accessible integer data object to be updated on the remote PE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>cond</td>
<td>cond is compared to the remote target value. If cond and the remote target are equal, then value is swapped into the remote target. Otherwise, the remote target is unchanged. In either case, the old value of the remote target is returned as the function return value. cond must be of the same data type as target.</td>
</tr>
<tr>
<td>IN</td>
<td>value</td>
<td>The value to be atomically written to the remote PE. value must be of the same data type as target.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>An integer that indicates the PE number upon which target is to be updated. If you are using Fortran, it must be a default integer value.</td>
</tr>
</tbody>
</table>

API description

The conditional swap routines conditionally update a target data object on an arbitrary PE and return the prior contents of the data object in one atomic operation.

The target and source data objects must conform to certain typing constraints, which are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_CSWAP</td>
<td>4-byte integer.</td>
</tr>
<tr>
<td>SHMEM_INT8_CSWAP</td>
<td>8-byte integer.</td>
</tr>
</tbody>
</table>

Return Values

The contents that had been in the target data object on the remote PE prior to the conditional swap. Data type is the same as the target data type.

Notes

None.

EXAMPLES

The following call ensures that the first PE to execute the conditional swap will successfully write its PE number to race_winner on PE 0.

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

int main(void)
{
    static int race_winner = -1;
```
int oldval;
start_pes(2);
oldval = shmem_int_cswap(&race_winner, -1, _my_pe(), 0);
if(oldval == -1) printf("pe %d was first\n",_my_pe());
return 1;
}

8.4.3 SHMEM_SWAP

Performs an atomic swap to a remote data object.

SYNOPSIS

C/C++:

do\nle shmem_double_swap(double *target, double value, int pe);
float shmem_float_swap(float *target, float value, int pe);
int shmem_int_swap(int *target, int value, int pe);
long shmem_long_swap(long *target, long value, int pe);
long long shmem_longlong_swap(long long *target, long long value, int pe);
long shmem_swap(long *target, long value, int pe);

FORTRAN:

INTEGER SHMEM_SWAP, value, pe
ires = SHMEM_SWAP(target, value, pe)
INTEGER*4 SHMEM_INT4_SWAP, value_i4, ires_i4
ires_i4 = SHMEM_INT4_SWAP(target, value_i4, pe)
INTEGER*8 SHMEM_INT8_SWAP, value_i8, ires_i8
ires_i8 = SHMEM_INT8_SWAP(target, value_i8, pe)
REAL*4 SHMEM_REAL4_SWAP, value_r4, res_r4
res_r4 = SHMEM_REAL4_SWAP(target, value_r4, pe)
REAL*8 SHMEM_REAL8_SWAP, value_r8, res_r8
res_r8 = SHMEM_REAL8_SWAP(target, value_r8, pe)

DESCRIPTION

Arguments

| OUT  | target | The remotely accessible integer data object to be updated on the remote PE. If you are using C/C++, the type of target should match that implied in the SYNOPSIS section. |
| IN   | value  | Value to be atomically written to the remote PE. value is the same type as target. |
| IN   | pe     | An integer that indicates the PE number on which target is to be updated. If you are using Fortran, it must be a default integer value. |

API description

shmem_swap performs an atomic swap operation. It writes value value into target on PE and returns the previous contents of target as an atomic operation.

If you are using Fortran, target must be of the following type:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_SWAP</td>
<td>Integer of default kind</td>
</tr>
</tbody>
</table>
8. OPENSHMEM LIBRARY API

<table>
<thead>
<tr>
<th>Library Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_SWAP</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_SWAP</td>
<td>8-byte integer</td>
</tr>
<tr>
<td>SHMEM_REAL4_SWAP</td>
<td>4-byte real</td>
</tr>
<tr>
<td>SHMEM_REAL8_SWAP</td>
<td>8-byte real</td>
</tr>
</tbody>
</table>

**Return Values**
The contents that had been at the target address on the remote PE prior to the swap is returned.

**Notes**
None.

**EXAMPLES**
The example below swap values between odd numbered PEs and their right (modulo) neighbor and outputs the result of swap.

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

int main(void)
{
    long *target;
    int me, npes;
    long swapped_val, new_val;

    start_pes(0);
    me = _my_pe();
    npes = _num_pes();
    target = (long *) shmalloc(sizeof(*target));
    *target = me;
    shmem_barrier_all();
    new_val = me;
    if (me & 1){
        swapped_val = shmem_long_swap(target, new_val, (me + 1) % npes);
        printf("%d: target = %ld, swapped = %ld\n", me, *target, swapped_val);
    }
    shfree(target);
    return 0;
}
```

8.4.4 SHMEM_FINC
Performs an atomic fetch-and-increment operation on a remote data object.

**SYNOPSIS**

**C/C++:**

```c
int shmem_int_finc(int *target, int pe);
long shmem_long_finc(long *target, int pe);
long long shmem_longlong_finc(long long *target, int pe);
```

**FORTRAN:**

```fortran
INTEGER pe
INTEGER*4 SHMEM_INT4_FINC, ires_i4
ires_i4 = SHMEM_INT4_FINC(target, pe)
INTEGER*8 SHMEM_INT8_FINC, ires_i8
ires_i8 = SHMEM_INT8_FINC(target, pe)
```
8. OPENSHMEM LIBRARY API

DESCRIPTION

Arguments

IN $target$ The remotely accessible integer data object to be updated on the remote PE. The type of $target$ should match that implied in the SYNOPSIS section.

IN $pe$ An integer that indicates the PE number on which $target$ is to be updated. If you are using Fortran, it must be a default integer value.

API description

These functions perform a fetch-and-increment operation. The $target$ on PE $pe$ is increased by one and the function returns the previous contents of $target$ as an atomic operation.

If you are using Fortran, $target$ must be of the following type:

<table>
<thead>
<tr>
<th>Routine Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_FINC 4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_FINC 8-byte integer</td>
</tr>
</tbody>
</table>

Return Values

The contents that had been at the $target$ address on the remote PE prior to the increment. The data type of the return value is the same as the $target$.

Notes

None.

EXAMPLES

The following $shmem_finc$ example is for C/C++ programs:

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

int dst;

int main(void)
{
    int me;
    int old;

    startpes(0);
    me = _my_pe();

    old = -1;
    dst = 22;
    shmem_barrier_all();

    if (me == 0)
        old = shmem_int_finc(&dst, 1);
    shmem_barrier_all();
    printf("%d: old = %d, dst = %d\n", me, old, dst);
    return 0;
}
```

8.4.5 SHMEM_INC

Performs an atomic increment operation on a remote data object.
SYNOPSIS

C/C++:

```c
void shmem_int_inc(int *target, int pe);
void shmem_long_inc(long *target, int pe);
void shmem_longlong_inc(long long *target, int pe);
```

FORTRAN:

```fortran
INTEGER pe
CALL SHMEM_INT4_INC(target, pe)
CALL SHMEM_INT8_INC(target, pe)
```

DESCRIPTION

Arguments

- **IN target**
  - The remotely accessible integer data object to be updated on the remote PE. The type of `target` should match that implied in the SYNOPSIS section.

- **IN pe**
  - An integer that indicates the PE number on which `target` is to be updated. If you are using Fortran, it must be a default integer value.

API description

These functions perform an atomic increment operation on the `target` data object on PE.

If you are using Fortran, `target` must be of the following type:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_INC</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_INC</td>
<td>8-byte integer</td>
</tr>
</tbody>
</table>

Return Values

None.

Notes

The term remotely accessible is defined in the Introduction.

EXAMPLES

The following `shmmem_int_inc` example is for C/C++ programs:

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

int dst;

int main(void)
{
  int me;
  start_pes(0);
  me = my_pe();
  dst = 74;
  shmem_barrier_all();
```
if (me == 0)
    shmem_int_inc(&dst, 1);
    shmem_barrier_all();
    printf("%d: dst = %d\n", me, dst);
    return 0;
}

8.4.6 SHMEM_FADD

Performs an atomic fetch-and-add operation on a remote data object.

SYNOPSIS

C/C++:

int shmem_int_fadd(int *target, int value, int pe);
long shmem_long_fadd(long *target, long value, int pe);
long long shmem_longlong_fadd(long long *target, long long value, int pe);

FORTRAN:

INTEGER pe
INTEGER*4 SHMEM_INT4_FADD, ires_i4, value_i4
ires_i4 = SHMEM_INT4_FADD(target, value_i4, pe)
INTEGER*8 SHMEM_INT8_FADD, ires_i8, value_i8
ires_i8 = SHMEM_INT8_FADD(target, value_i8, pe)

DESCRIPTION

Arguments

OUT target The remotely accessible integer data object to be updated on the remote PE. The type of target should match that implied in the SYNOPSIS section.

IN value The value to be atomically added to target. The type of value should match that implied in the SYNOPSIS section.

IN pe An integer that indicates the PE number on which target is to be updated. If you are using Fortran, it must be a default integer value.

API description

shmem_fadd functions perform an atomic fetch-and-add operation. An atomic fetch-and-add operation fetches the old target and adds value to target without the possibility of another atomic operation on the target between the time of the fetch and the update. These routines add value to target on pe and return the previous contents of target as an atomic operation.

If you are using Fortran, target must be of the following type:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_FADD</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_FADD</td>
<td>8-byte integer</td>
</tr>
</tbody>
</table>

Return Values

The contents that had been at the target address on the remote PE prior to the atomic addition operation. The data type of return value is the same as the target.
8. OPENSHMEM LIBRARY API

Notes
None.

EXAMPLES

The following `shm_fadd` example is for C/C++ programs:

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

int main(void)
{
    int me, old;
    static int dst;
    start_pes(0);
    me = _my_pe();
    old = -1;
    dst = 22;
    shmem_barrier_all();
    if (me == 1){
        old = shmem_int_fadd(&dst, 44, 0);
    }
    shmem_barrier_all();
    printf("%d: old = %d, dst = %d\n", me, old, dst);
    return 0;
}
```

8.5 Collective Operations

Collective operations are defined as communication or synchronization operations on a group of PEs called `Active set`. The collective operations require all PEs in the `Active set` to simultaneously call the operation. A PE that is not part of the `Active set` calling the collective operations results in an undefined behavior. All collective operations have an `Active set` as an input parameter except `SHMEM_BARRIER_ALL`. The `SHMEM_BARRIER_ALL` is called by all PEs of the OpenSHMEM program.

The `Active set` is defined by the arguments `PE_start`, `logPE_stride`, and `PE_size`. `PE_start` is the starting PE number, a log (base 2) of `logPE_stride` is the stride between PEs, and `PE_size` is the number of PEs participating in the `Active set`. All PEs participating in the collective operations provide the same values for these arguments.

Another argument important to collective operations is `pSync`, which is a symmetric work array. All PEs participating in a collective must pass the same `pSync` array. On completion of a collective call, the `pSync` is restored to its original contents. The reuse of `pSync` array is allowed for a PE, if all previous collective operations using the `pSync` array have been completed by all participating PEs. One can use a synchronization collective operation such as `SHMEM_BARRIER` to ensure completion of previous collective operations. The `shmem_barrier` function allows the same `pSync` array to be used on consecutive calls as long as the active PE set does not change.

All collective operations defined in the specification are blocking. The collective operations return on completion. The collective operations defined in the OpenSHMEM specification are:

- `SHMEM_BROADCAST`
- `SHMEM_BARRIER`
- `SHMEM_BARRIER_ALL`
- `SHMEM_COLLECT`
8.5.1 SHMEM_BARRIER_ALL

Registers the arrival of a PE at a barrier and suspends PE execution until all other PEs arrive at the barrier and all local and remote memory updates are completed.

SYNOPSIS

C/C++:

```c
void shmem_barrier_all(void);
```

FORTRAN:

```fortran
CALL SHMEM_BARRIER_ALL
```

DESCRIPTION

Arguments

None.

API description

The `shmem_barrier_all` function registers the arrival of a PE at a barrier. Barriers are a fast mechanism for synchronizing all PEs at once. This routine causes a PE to suspend execution until all PEs have called `shmem_barrier_all`. This function must be used with PEs started by `start_pes`.

Prior to synchronizing with other PEs, `shmem_barrier_all` ensures completion of all previously issued memory stores and remote memory updates issued via OpenSHMEM AMOs and RMA routine calls such as `shmem_int_add` and `shmem_put32`.

Return Values

None.

Notes

None.

EXAMPLES

The following `shmem_barrier_all` example is for C/C++ programs:

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

int x=1010;

int main(void)
{
    int me, npes;
    start_pes(0);
    me = _my_pe();
    npes = _num_pes();

    /*put to next PE in a circular fashion*/
    shmem_int_p(&x, 4, (me+1)%npes);
    /*synchronize all PEs*/
    shmem_barrier_all();

    printf("%d: x = %d\n", me, x);
    return 0;
}
```
8.5.2 SHMEM_BARRIER

Performs all operations described in the shmem_barrier_all interface but with respect to a subset of PEs defined by the Active set.

SYNOPSIS

C/C++:

```c
void shmem_barrier(int PE_start, int logPE_stride, int PE_size, long *pSync);
```

FORTRAN:

```fortran
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BARRIER_SYNC_SIZE)
CALL SHMEM_BARRIER(PE_start, logPE_stride, PE_size, pSync)
```

DESCRIPTION

Arguments

- **IN PE_start**  The lowest virtual PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value.
- **IN logPE_stride**  The log (base 2) of the stride between consecutive virtual PE numbers in the Active set. logPE_stride must be of type integer. If you are using Fortran, it must be a default integer value.
- **IN PE_size**  The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.
- **IN pSync**  A symmetric work array. In C/C++, pSync must be of type long and size SHMEM_BARRIER_SYNC_SIZE. In Fortran, pSync must be of type integer and size SHMEM_BARRIER_SYNC_SIZE. If you are using Fortran, it must be a default integer type. Every element of this array must be initialized to SHMEM_SYNC_VALUE before any of the PEs in the Active set enter shmem_barrier the first time.

API description

shmem_barrier is a collective synchronization routine over an Active set. Control returns from shmem_barrier after all PEs in the Active set (specified by PE_start, logPE_stride, and PE_size) have called shmem_barrier. As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the Active set call the routine. If a PE not in the Active set calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments PE_start, logPE_stride, and PE_size must be equal on all PEs in the Active set. The same work array must be passed in pSync to all PEs in the Active set.

shmem_barrier ensures that all previously issued stores and remote memory updates, including AMOs and RMA operations, done by any of the PEs in the Active set are complete before returning.

The same pSync array may be reused on consecutive calls to shmem_barrier if the same active PE set is used.

Return Values

None.

Notes

If the pSync array is initialized at run time, be sure to use some type of synchronization, for example, a call to shmem_barrier_all, before calling shmem_barrier for the first time.

If the Active set does not change, shmem_barrier can be called repeatedly with the same pSync array. No additional synchronization beyond that implied by shmem_barrier itself is necessary in this case.
8. OPENSHMEM LIBRARY API

EXAMPLES

The following barrier example is for C/C++ programs:

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

long pSync[_SHMEM_BARRIER_SYNC_SIZE];
int x = 10101;

int main(void)
{
    int i, me, npes;

    for (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*/
        shmem_int_p(&x, 4, (me+2)%npes);
        /*synchronize all even pes*/
        shmem_barrier(0, 1, (npes/2 + npes%2), pSync);
    }
    printf("%d: x = %d\n", me, x);
    return 0;
}
```

8.5.3 SHMEM_BROADCAST

Broadcasts a block of data from one PE to one or more target PEs.

SYNOPSIS

C/C++:

```c
void shmem_broadcast32(void *target, const void *source, size_t nlong, int PE_root, int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_broadcast64(void *target, const void *source, size_t nlong, int PE_root, int PE_start, int logPE_stride, int PE_size, long *pSync);
```

FORTRAN:

```fortran
INTEGER nlong, PE_root, PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
CALL SHMEM_BROADCAST4(target, source, nlong, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST8(target, source, nlong, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST32(target, source, nlong, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST64(target, source, nlong, PE_root, PE_start, logPE_stride, PE_size, pSync)
```

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>OUT</th>
<th>target</th>
<th>A symmetric data object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>source</td>
<td>A symmetric data object that can be of any data type that is permissible for the target argument.</td>
</tr>
</tbody>
</table>
The number of elements in source. For shmem_broadcast32 and shmem_broadcast4, this is the number of 32-bit halfwords. nlong must be of type size_t in C. If you are using Fortran, it must be a default integer value.

Zero-based ordinal of the PE, with respect to the Active set, from which the data is copied. Must be greater than or equal to 0 and less than PE_size. PE_root must be of type integer. If you are using Fortran, it must be a default integer value.

The lowest virtual PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value.

The log (base 2) of the stride between consecutive virtual PE numbers in the Active set. log_PE_stride must be of type integer. If you are using Fortran, it must be a default integer value.

The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.

A symmetric work array. In C/C++, pSync must be of type long and \_SHMEM\_BCAST\_SYNC\_SIZE. In Fortran, pSync must be of type integer and size SHMEM\_BCAST\_SYNC\_SIZE. Every element of this array must be initialized with the value \_SHMEM\_SYNC\_VALUE (in C/C++) or SHMEM\_SYNC\_VALUE (in Fortran) before any of the PEs in the Active set enter shmem\_barrier.

OpenSHMEM broadcast routines are collective routines. They copy data object source on the processor specified by PE_root and store the values at target on the other PEs specified by the triplet PE_start, logPE_stride, PE_size. The data is not copied to the target area on the root PE.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the Active set call the routine. If a PE not in the Active set calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments PE_root, PE_start, logPE_stride, and PE_size must be equal on all PEs in the Active set. The same target and source data objects and the same pSync work array must be passed to all PEs in the Active set.

Before any PE calls a broadcast routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this): The pSync array on all PEs in the Active set is not still in use from a prior call to a broadcast routine. The target array on all PEs in the Active set is ready to accept the broadcast data.

Upon return from a broadcast routine, the following are true for the local PE: If the current PE is not the root PE, the target data object is updated. The values in the pSync array are restored to the original values.

The target and source data objects must conform to certain typing constraints, which are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type of target and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_broadcast8,</td>
<td>Any noncharacter type that has an element size of 64 bits. No Fortran derived types or C/C++ structures are allowed.</td>
</tr>
<tr>
<td>shmem_broadcast64</td>
<td>Any noncharacter type that has an element size of 32 bits. No Fortran derived types or C/C++ structures are allowed.</td>
</tr>
<tr>
<td>shmem_broadcast32</td>
<td>Any noncharacter type that has an element size of 32 bits. No Fortran derived types or C/C++ structures are allowed.</td>
</tr>
<tr>
<td>shmem_broadcast4</td>
<td>Any noncharacter type that has an element size of 32 bits. No Fortran derived types or C/C++ structures are allowed.</td>
</tr>
</tbody>
</table>

Return Values

None.
Notes
All OpenSHMEM broadcast routines restore $pSync$ to its original contents. Multiple calls to OpenSHMEM routines that use the same $pSync$ array do not require that $pSync$ be reinitialized after the first call.
You must ensure that the $pSync$ array is not being updated by any PE in the Active set while any of the PEs participates in processing of an OpenSHMEM broadcast routine. Be careful to avoid these situations:
If the $pSync$ array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized $pSync$ before any of them enter an OpenSHMEM routine called with the $pSync$ synchronization array. A $pSync$ array may be reused on a subsequent OpenSHMEM broadcast routine only if none of the PEs in the Active set are still processing a prior OpenSHMEM broadcast routine call that used the same $pSync$ array. In general, this can be ensured only by doing some type of synchronization.

EXAMPLES
In the following examples, the call to `shmem_broadcast64` copies `source` on PE 4 to `target` on PEs 5, 6, and 7.

C/C++ example:
```c
#include <stdio.h>
#include <stdlib.h>
#include <shmem.h>
#define NUM_ELEMS 4
long pSync[_SHMEM_BCAST_SYNC_SIZE];
long source[NUM_ELEMS], target[NUM_ELEMS];

int main(void)
{
    int i, me, npes;
    start_pes(0);
    me = _my_pe();
    npes = _num_pes();

    if (me == 0)
        for (i = 0; i < NUM_ELEMS; i++)
            source[i] = i;
    for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++)
        pSync[i] = _SHMEM_SYNC_VALUE;
    shmem_barrier_all(); /* Wait for all PEs to initialize pSync */
    shmem_broadcast64(target, source, NUM_ELEMS, 0, 0, 0, npes, pSync);
    printf("%ld", me, target[0]);
    for (i = 1; i < NUM_ELEMS; i++)
        printf("%ld", target[i]);
    printf("\n");
    return 0;
}
```

Fortran example:
```fortran
INTEGER PSYNC(_SHMEM_BCAST_SYNC_SIZE)
INTEGER TARGET, SOURCE, NLONG, PE_ROOT, PE_START,
& LOGPE_STRIDE, PE_SIZE, PSYNC
COMMON /COM/ TARGET, SOURCE

DATA PSYNC /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/

CALL SHMEM_BROADCAST64(TARGET, SOURCE, NLONG, 0, 4, 0, 4, PSYNC)
```
8.5.4 SHMEM_COLLECT, SHMEM_FCOLLECT

Concatenates blocks of data from multiple PEs to an array in every PE.

SYNOPSIS

C/C++:

```c
void shmem_collect32(void *target, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_collect64(void *target, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_fcollect32(void *target, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_fcollect64(void *target, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
```

FORTRAN:

```fortran
INTEGER nelems
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)
CALL SHMEM_COLLECT4(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT8(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT32(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT64(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT4(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT8(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT32(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT64(target, source, nelems, PE_start, logPE_stride, PE_size, pSync)
```

DESCRIPTION

Arguments

OUT  `target`  A symmetric array. The `target` argument must be large enough to accept the concatenation of the `source` arrays on all PEs. The data types are as follows: For `shmem_collect8`, `shmem_collect64`, `shmem_fcollect8`, and `shmem_fcollect64`, any data type with an element size of 64 bits. `Fortran` derived types, `Fortran` character type, and `C/C++` structures are not permitted. For `shmem_collect4`, `shmem_collect32`, `shmem_fcollect4`, and `shmem_fcollect32`, any data type with an element size of 32 bits. `Fortran` derived types, `Fortran` character type, and `C/C++` structures are not permitted.

IN  `source`  A symmetric data object that can be of any type permissible for the `target` argument.

IN  `nelems`  The number of elements in the `source` array. `nelems` must be of type `size_t` for `C`. If you are using `Fortran`, it must be a default integer value.

IN  `PE_start`  The lowest virtual PE number of the `Active set` of PEs. `PE_start` must be of type integer. If you are using `Fortran`, it must be a default integer value.

IN  `logPE_stride`  The log (base 2) of the stride between consecutive virtual PE numbers in the `Active set`. `logPE_stride` must be of type integer. If you are using `Fortran`, it must be a default integer value.

IN  `PE_size`  The number of PEs in the `Active set`. `PE_size` must be of type integer. If you are using `Fortran`, it must be a default integer value.
IN \quad pSync \quad A \ symmetric \ work \ array. \ In \ C/C++, \ pSync \ must \ be \ of \ type \ long \ and \ size \ _SHMEM_COLLECT_SYNC_SIZE. \ In \ Fortran, \ pSync \ must \ be \ of \ type \ integer \ and \ size \ SHMEM_COLLECT_SYNC_SIZE. \ If \ you \ are \ using \ Fortran, \ it \ must \ be \ a \ default \ integer \ value. \ Every \ element \ of \ this \ array \ must \ be \ initialized \ with \ the \ value \ _SHMEM_SYNC_VALUE \ in \ C/C++ \ or \ SHMEM_SYNC_VALUE \ in \ Fortran \ before \ any \ of \ the \ PEs \ in \ the \ Active \ set \ enter \ shmem_barrier.

API description

OpenSHMEM collect and fcoll routines concatenate nelems 64-bit or 32-bit data items from the source array into the target array, over the set of PEs defined by PE_start, log2PE_stride, and PE_size, in processor number order. The resultant target array contains the contribution from PE PE_start first, then the contribution from PE PE_start + PE_stride second, and so on. The collected result is written to the target array for all PEs in the Active set.

The fcoll routines require that nelems be the same value in all participating PEs, while the collect routines allow nelems to vary from PE to PE.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the Active set call the routine. If a PE not in the Active set and calls this collective routine, the behavior is undefined.

The values of arguments PE_start, log2PE_stride, and PE_size must be equal on all PEs in the Active set. The same target and source arrays and the same pSync work array must be passed to all PEs in the Active set.

Upon return from a collective routine, the following are true for the local PE: The target array is updated. The values in the pSync array are restored to the original values.

Return Values

None.

Notes

All OpenSHMEM collective routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first time it is used.

You must ensure that the pSync array is not being updated on any PE in the Active set while any of the PEs participate in processing of an OpenSHMEM collective routine. Be careful to avoid these situations: If the pSync array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized pSync before any of them enter an OpenSHMEM routine called with the pSync synchronization array. A pSync array can be reused on a subsequent OpenSHMEM collective routine only if none of the PEs in the Active set are still processing a prior OpenSHMEM collective routine call that used the same pSync array. In general, this may be ensured only by doing some type of synchronization.

The collective routines operate on active PE sets that have a non-power-of-two PE_size with some performance degradation. They operate with no performance degradation when nelems is a non-power-of-two value.

EXAMPLES

The following shmem_collect example is for C/C++ programs:

```c
#include <stdio.h>
#include <stdlib.h>
#include <shmem.h>

long pSync[_SHMEM_COLLECT_SYNC_SIZE];
int source[2];

int main(void)
{
    int i, me, npes;
    int *target;
```
8. OPENSHMEM LIBRARY API

```c
start_pes(0);
me = __my_pe();
npes = __num_pes();

source[0] = me * 2;
source[1] = me * 2 + 1;
target = (int *) shmalloc(sizeof(int) * npes * 2);
for (i=0; i < _SHMEM_COLLECT_SYNC_SIZE; i++) {
pSync[i] = _SHMEM_SYNC_VALUE;
}
shm_barrier_all(); /* Wait for all PEs to initialize pSync */
shmem_collect32(target, source, 2, 0, norp, pSync);
for (i = 0; i < npes * 2; i++)
    printf("\n");
return 0;
```

The following SHMEM_COLLECT example is for Fortran programs:

```fortran
INTEGER PSYNC(SHMEM_COLLECT_SYNC_SIZE)
DATA PSYNC /SHMEM_COLLECT_SYNC_SIZE*SHMEM_SYNC_VALUE/
CALL SHMEM_COLLECT4(TARGET, SOURCE, 64, PE_START, LOGPE_STRIDE,
& PE_SIZE, PSYNC)
```

### 8.5.5 SHMEM_REDUCIONS

Performs a logical operations across a set of PEs.

**SYNOPSIS**

C/C++:

```c
void shmem_int_and_to_all(int *target, int *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_and_to_all(long *target, long *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_and_to_all(long long *target, long long *source, int nreduce, int
  PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_and_to_all(short *target, short *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_double_max_to_all(double *target, double *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_max_to_all(float *target, float *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_max_to_all(int *target, int *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_max_to_all(long *target, long *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_max_to_all(long double *target, long double *source, int nreduce, int
  PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_max_to_all(long long *target, long long *source, int nreduce, int
  PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_max_to_all(short *target, short *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_double_min_to_all(double *target, double *source, int nreduce, int PE_start, int
  logPE_stride, int PE_size, double *pWrk, long *pSync);
```

8. OPENSHMEM LIBRARY API

```c
void shmem_float_min_to_all(float *target, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *target, long long *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_min_to_all(short *target, short *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_complexd_min_to_all(complex *target, double complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
void shmem_complexf_min_to_all(float complex *target, float complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
void shmem_double_min_to_all(double *target, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *target, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *target, long long *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_complexd_min_to_all(complex *target, double complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
void shmem_complexf_min_to_all(float complex *target, float complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
void shmem_double_min_to_all(double *target, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *target, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *target, long long *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_complexd_min_to_all(complex *target, double complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
void shmem_complexf_min_to_all(float complex *target, float complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
void shmem_double_min_to_all(double *target, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *target, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *target, long long *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_complexd_min_to_all(complex *target, double complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
void shmem_complexf_min_to_all(float complex *target, float complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
void shmem_double_min_to_all(double *target, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *target, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *target, long long *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_complexd_min_to_all(complex *target, double complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
void shmem_complexf_min_to_all(float complex *target, float complex *source, int nreduce,
int PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
void shmem_double_min_to_all(double *target, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *target, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *target, long long *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```
logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_longxor_to_all(long *target, long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlongxor_to_all(long long *target, long long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_shortxor_to_all(short *target, short *source, int nreduce, int PE_start, int logPE_stride, int PE_size, short *pWrk, long *pSync);

FORTRAN:
CALL SHMEM_INT4_AND_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_AND_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_MAX_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_MAX_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL4_MAX_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL8_MAX_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL16_MAX_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_MIN_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_MIN_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL4_MIN_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL8_MIN_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL16_MIN_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_COMP4_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_COMP8_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL4_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL8_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL16_SUM_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_COMP4_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_COMP8_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL4_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL8_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_REAL16_PROD_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_OR_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_OR_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_XOR_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_XOR_TO_ALL(target, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)

DESCRIPTION

Arguments

IN target A symmetric array, of length nreduce elements, to receive the result of the reduction operations. The data type of target varies with the version of the reduction routine being called. When calling from C/C++, refer to the SYNOPSIS section for data type information.

IN source A symmetric array, of length nreduce elements, that contains one element for each separate reduction operation. The source argument must have the same data type as target.

IN nreduce The number of elements in the target and source arrays. nreduce must be of type integer. If you are using Fortran, it must be a default integer value.

IN PE_start The lowest virtual PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value.

IN logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the Active set. logPE_stride must be of type integer. If you are using Fortran, it must be a default integer value.

IN PE_size The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.

IN pWrk A symmetric work array. The pWrk argument must have the same data type as target. In C/C++, this contains max(nreduce/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements. In Fortran, this contains max(nreduce/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements.

IN pSync A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. In Fortran, pSync must be of type integer and size _SHMEM_REDUCE_SYNC_SIZE. If you are using Fortran, it must be a default integer value. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE (in C/C++) or _SHMEM_SYNC_VALUE (in Fortran) before any of the PEs in the Active set enter the reduction routine.

API description

OpenSHMEM reduction routines compute one or more reductions across symmetric arrays on multiple
virtual PEs. A reduction performs an associative binary operation across a set of values.

The `nreduce` argument determines the number of separate reductions to perform. The `source` array on all PEs in the `Active set` provides one element for each reduction. The results of the reductions are placed in the `target` array on all PEs in the `Active set`. The `Active set` is defined by the `PE_start`, `logPE_stride`, `PE_size` triplet.

The `source` and `target` arrays may be the same array, but they may not be overlapping arrays.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the `Active set` call the routine. If a PE not in the `Active set` calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments `nreduce`, `PE_start`, `logPE_stride`, and `PE_size` must be equal on all PEs in the `Active set`. The same `target` and `source` arrays, and the same `pWrk` and `pSync` work arrays, must be passed to all PEs in the `Active set`.

Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a `barrier` or some other method is often needed to ensure this): The `pWrk` and `pSync` arrays on all PEs in the `Active set` are not still in use from a prior call to a collective OpenSHMEM routine. The `target` array on all PEs in the `Active set` is ready to accept the results of the reduction.

Upon return from a reduction routine, the following are true for the local PE: The `target` array is updated. The values in the `pSync` array are restored to the original values.

When calling from Fortran, the `target` data types are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shmem_int8_and_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_int4_and_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_comp8_max_to_all</code></td>
<td>Complex, with an element size equal to two 8-byte real values.</td>
</tr>
<tr>
<td><code>shmem_int4_max_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_int8_max_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_real4_max_to_all</code></td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_real16_max_to_all</code></td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
<tr>
<td><code>shmem_int4_min_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_int8_min_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_real4_min_to_all</code></td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_real16_min_to_all</code></td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
<tr>
<td><code>shmem_comp4_sum_to_all</code></td>
<td>Complex, with an element size equal to two 4-byte real values.</td>
</tr>
<tr>
<td><code>shmem_comp8_sum_to_all</code></td>
<td>Complex, with an element size equal to two 8-byte real values.</td>
</tr>
<tr>
<td><code>shmem_int4_sum_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_int8_sum_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_real4_sum_to_all</code></td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_real16_sum_to_all</code></td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
<tr>
<td><code>shmem_comp4_prod_to_all</code></td>
<td>Complex, with an element size equal to two 4-byte real values.</td>
</tr>
<tr>
<td><code>shmem_comp8_prod_to_all</code></td>
<td>Complex, with an element size equal to two 8-byte real values.</td>
</tr>
<tr>
<td><code>shmem_int4_prod_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_int8_prod_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_real4_prod_to_all</code></td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_real8_prod_to_all</code></td>
<td>Real, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_real16_prod_to_all</code></td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
<tr>
<td><code>shmem_int8_or_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_int4_or_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td><code>shmem_int8_xor_to_all</code></td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td><code>shmem_int4_xor_to_all</code></td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
</tbody>
</table>
Return Values

None.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first time it is used.

You must ensure that the pSync array is not being updated on any PE in the Active set while any of the PEs participate in processing of an OpenSHMEM reduction routine. Be careful to avoid the following situations: If the pSync array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized pSync before any of them enter an OpenSHMEM routine called with the pSync synchronization array. A pSync or pWrk array can be reused in a subsequent reduction routine call only if none of the PEs in the Active set are still processing a prior reduction routine call that used the same pSync or pWrk arrays. In general, this can be assured only by doing some type of synchronization.

EXAMPLES

This Fortran reduction example statically initializes the pSync array and finds the logical AND of the integer variable FOO across all even PEs.

```
INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
INTEGER*4 PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
INTEGER FOO, FOOAND
SAVE FOO, FOOAND, PWRK
INTRINSIC MY_PE

FOO = MY_PE()
IF ( MOD(MY_PE(),2) .EQ. 0) THEN
  IF ( MOD(NUM_PES(),2) .EQ. 0) THEN
    CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2, &
      PWRK, PSYNC)
  ELSE
    CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2+1, &
      PWRK, PSYNC)
  ENDIF
  PRINT*, 'Result on PE ',MY_PE(),' is ',FOOAND
ENDIF
```

This Fortran example statically initializes the pSync array and finds the maximum value of real variable FOO across all even PEs.

```
INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOMAX, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOMAX, PWRK
INTRINSIC MY_PE

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
  CALL SHMEM_REAL8_MAX_TO_ALL(FOOMAX, FOO, NR, 0, 1, NPES/2, &
    PWRK, PSYNC)
  PRINT*, 'Result on PE ',MY_PE(),' is ', FOOMAX
ENDIF
```

This Fortran example statically initializes the pSync array and finds the minimum value of real variable FOO across all the even PEs.

```
INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOMAX, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOMAX, PWRK
INTRINSIC MY_PE

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
  CALL SHMEM_REAL8_MIN_TO_ALL(FOOMAX, FOO, NR, 0, 1, NPES/2, &
    PWRK, PSYNC)
  PRINT*, 'Result on PE ',MY_PE(),' is ', FOOMAX
ENDIF
```
This Fortran example statically initializes the pSync array and finds the sum of the real variable FOO across all even PEs.

This Fortran example statically initializes the pSync array and finds the product of the real variable FOO across all the even PEs.

This Fortran example statically initializes the pSync array and finds the logical OR of the integer variable FOO across all even PEs.
This Fortran example statically initializes the pSync array and computes the exclusive XOR of variable FOO across all even PEs.

```
INCLUDE "mpp/shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOXOR, PWKR (MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOXOR, PWKR
INTRINSIC MY_PE

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
  CALL SHMEM_REAL8_XOR_TO_ALL(FOOXOR, FOO, NR, 0, 1, N$PES/2,
                             & PWKR, PSYNC)
  PRINT*, 'Result on PE ',MY_PE(),' is ',FOOXOR
ENDIF
```

## 8.6 Point-To-Point Synchronization Functions

The following section discusses OpenSHMEM API that provides a mechanism for synchronization between two PEs based on the value of a symmetric data object.

### 8.6.1 SHMEM_WAIT

Wait for a variable on the local PE to change.

**SYNOPSIS**

**C/C++:**

```c
void shmem_int_wait(int *ivar, int cmp_value);
void shmem_int_wait_until(int *ivar, int cmp, int cmp_value);
void shmem_long_wait(long *ivar, long cmp_value);
void shmem_long_wait_until(long *ivar, int cmp, long cmp_value);
void shmem_longlong_wait_until(long long *ivar, int cmp, long long cmp_value);
void shmem_short_wait(short *ivar, short cmp_value);
void shmem_short_wait_until(short *ivar, int cmp, short cmp_value);
void shmem_wait(long *ivar, long cmp_value);
void shmem_wait_until(long *ivar, int cmp, long cmp_value);
```

**FORTRAN:**

```fortran
CALL SHMEM_INT4_WAIT(ivar, cmp_value)
CALL SHMEM_INT4_WAIT_UNTIL(ivar, cmp, cmp_value)
CALL SHMEM_INT8_WAIT(ivar, cmp_value)
CALL SHMEM_WAIT(ivar, cmp_value)
CALL SHMEM_WAIT_UNTIL(ivar, cmp, cmp_value)
```

**DESCRIPTION**

**Arguments**

- **OUT** `ivar`  
  A remotely accessible integer variable that is being updated by another PE. If you are using C/C++, the type of ivar should match that implied in the SYNOPSIS section.
IN cmp The compare operator that compares ivar with cmp_value. cmp must be of type integer. If you are using Fortran, it must be of default kind. If you are using C/C++, the type of cmp should match that implied in the SYNOPSIS section.

IN cmp_value cmp_value must be of type integer. If you are using C/C++, the type of cmp_value should match that implied in the SYNOPSIS section. If you are using Fortran, cmp_value must be an integer of the same size and kind as ivar.

API description

shmem_wait and shmem_wait_until wait for ivar to be changed by a remote write or atomic swap issued by a different processor. These routines can be used for point-to-point directed synchronization. A call to shmem_wait does not return until some other processor writes a value, not equal to cmp_value, into ivar on the waiting processor. A call to shmem_wait_until does not return until some other processor changes ivar to satisfy the condition implied by cmp and cmp_value. This mechanism is useful when a processor needs to tell another processor that it has completed some action. The shmem_wait routines return when ivar is no longer equal to cmp_value. The shmem_wait_until routines return when the compare condition is true. The compare condition is defined by the ivar argument compared with the cmp_value using the comparison operator, cmp.

If you are using Fortran, ivar must be a specific sized integer type according to the function being called, as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Type of ivar</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_wait, shmem_wait_until</td>
<td>default INTEGER</td>
</tr>
<tr>
<td>shmem_int4_wait, shmem_int4_wait_until</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>shmem_int8_wait, shmem_int8_wait_until</td>
<td>INTEGER*8</td>
</tr>
</tbody>
</table>

The following cmp values are supported:

<table>
<thead>
<tr>
<th>CMP Value</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SHMEM_CMP_EQ</td>
<td>Equal</td>
</tr>
<tr>
<td>_SHMEM_CMP_NE</td>
<td>Not equal</td>
</tr>
<tr>
<td>_SHMEM_CMP_GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>_SHMEM_CMP_LE</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>_SHMEM_CMP_LT</td>
<td>Less than</td>
</tr>
<tr>
<td>_SHMEM_CMP_GE</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

For Fortran:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_CMP_EQ</td>
<td>Equal</td>
</tr>
<tr>
<td>SHMEM_CMP_NE</td>
<td>Not equal</td>
</tr>
<tr>
<td>SHMEM_CMP_GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>SHMEM_CMP_LE</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>SHMEM_CMP_LT</td>
<td>Less than</td>
</tr>
<tr>
<td>SHMEM_CMP_GE</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

Return Values

None.

Notes
None.

EXAMPLES

The following call returns when variable ivar is not equal to 100:

```c
INTEGER*8 IVAR
CALL SHMEM_INT8_WAIT(IVAR, INTEGER*8(100))
```

The following call to `SHMEM_INT8_WAIT_UNTIL` is equivalent to the call to `SHMEM_INT8_WAIT` in example 1:

```c
INTEGER*8 IVAR
CALL SHMEM_INT8_WAIT_UNTIL(IVAR, SHMEM_CMP_NE, INTEGER*8(100))
```

The following C/C++ call waits until the sign bit in ivar is set by a transfer from a remote PE:

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

int ivar;
shmem_int_wait_until(&ivar, SHMEM_CMP_LT, 0);
```

The following Fortran example is in the context of a subroutine:

```fortran
SUBROUTINE EXAMPLE()
INTEGER FLAG_VAR
COMMON/FLAG/FLAG_VAR
.
. .
FLAG_VAR = FLAG_VALUE ! initialize the event variable
.
. .
IF (FLAG_VAR .EQ. FLAG_VALUE) THEN
  CALL SHMEM_WAIT(FLAG_VAR, FLAG_VALUE)
ENDIF
FLAG_VAR = FLAG_VALUE ! reset the event variable for next time
.
. .
END
```

8.7 Memory Ordering Operations

The following section discusses OpenSHMEM APIs that provide mechanisms to ensure ordering and/or delivery of `Put`, AMO, and memory store operations to symmetric data objects.

8.7.1 SHMEM_FENCE

Assures ordering of delivery of `Put`, AMOs, and memory store operations to symmetric data objects.

SYNOPSIS

C/C++:

```c
void shmem_fence(void);
```

FORTRAN:

```fortran
CALL shmem_fence
```

DESCRIPTION

Arguments

None.
API description
This function assures ordering of delivery of Put, AMOs, and memory store operations to symmetric data objects. All Put, AMOs, and memory store operations to symmetric data objects issued to a particular remote PE prior to the call to shmem_fence are guaranteed to be delivered before any subsequent Put, AMOs, and memory store operations to symmetric data objects to the same PE. shmem_fence guarantees order of delivery, not completion.

Return Values
None.

Notes
shmem_fence only provides per-PE ordering guarantees and does not guarantee completion of delivery. There is a subtle difference between shmem_fence and shmem_quiet, in that, shmem_quiet guarantees completion of Put, AMOs, and memory store operations to symmetric data objects which makes the updates visible to all other PEs.

The shmem_quiet function should be called if completion of PUT, AMOs, and memory store operations to symmetric data objects is desired when multiple remote PEs are involved.

EXAMPLES
The following shmem_fence example is for C/C++ programs:
```c
#include <stdio.h>
#include <shmem.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 %ld\n", _my_pe(), target[0]);
    return 1;
}
```

Put1 will be ordered to be delivered before put3 and put2 will be ordered to be delivered before put4.

8.7.2 SHMEM_QUiet

Waits for completion of all outstanding Put, AMOs and memory store operations to symmetric data objects issued by a PE.

SYNOPSIS

C/C++:
```c
void shmem_quiet(void);
```

FORTRAN:
CALL SHMEM_QUIET

DESCRIPTION

Arguments
None.

API description
The shmem_quiet routine ensures completion of Put, AMOs, and memory store operations on symmetric data issued by the calling PE. All Put, AMOs, memory store operations to symmetric data objects are guaranteed to be completed and visible to all PEs when shmem_quiet returns.

Return Values
None.

Notes
shmem_quiet is most useful as a way of ensuring completion of several Put, AMOs, and memory store operations to symmetric data objects initiated by the calling PE. For example, you might use shmem_quiet to await delivery of a block of data before issuing another Put, which sets a completion flag on another PE. shmem_quiet is not usually needed if shmem_barrier_all or shmem_barrier are called. The barrier routines wait for the completion of outstanding writes (Put, AMO, memory stores) to symmetric data objects on all PEs.

EXAMPLES

The following example uses shmem_quiet in a C/C++ program:

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

long target[3] = {0};
int targ = 0;
long source[3] = {1, 2, 3};
int src = 90;

int main(void)
{
    long x[3] = {0};
    int y = 0;

    start_pes(0);
    if (_my_pe() == 0) {
        shmem_long_put(target, source, 3, 1); /*put1*/
        shmem_int_put(&targ, &src, 1, 2); /*put2*/
        shmem_quiet();
        shmem_long_get(x, target, 3, 1); /*gets updated value from target on PE 1 to local array x*/
        shmem_int_get(&y, &targ, 1, 2); /*gets updated value from targ on PE 2 to local variable y*/
        printf("x: {%ld,%ld,%ld}\n",x[0],x[1],x[2]); /*x: {1,2,3}*/
        printf("y: %d\n", y); /*y: 90*/
        shmem_int_put(&targ, &src, 1, 1); /*put3*/
        shmem_int_put(&targ, &src, 1, 2); /*put4*/
    }
    shmem_barrier_all(); /* sync sender and receiver */
    return 0;
}
```
Put1 and put2 will be completed and visible before put3 and put4.

### 8.7.3 Synchronization and Communication Ordering in OpenSHMEM

When using the OpenSHMEM API, synchronization, ordering, and completion of communication become critical. The updates via Put operations, AMOs and store operations on symmetric data cannot be guaranteed until some form of synchronization or ordering is introduced by the application programmer. The table below gives the different synchronization and ordering choices, and the situations where they may be useful.

<table>
<thead>
<tr>
<th>OpenSHMEM API</th>
<th>Working of OpenSHMEM API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-point synchronization shmem_wait, shmem_wait_until</td>
<td><img src="image_url" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Waits for a symmetric variable to be updated by a remote PE. Should be used when computation on the local PE cannot proceed without the value that the remote PE is to update.

| Ordering puts issued by a local PE shmem_fence | ![Diagram](image_url) |

All Put operations, AMOs and store operations on symmetric data issued to same PE are guaranteed to be delivered before Puts (to the same PE) issued after the fence call.
### OpenSHMEM API

<table>
<thead>
<tr>
<th>Working of OpenSHMEM API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ordering puts issued by all PE</strong></td>
</tr>
<tr>
<td><strong>shmem_quiet</strong></td>
</tr>
</tbody>
</table>

**Diagram:**

- **PE 0**
  - `shmem_int_p(addr1, value1, PE 0)`
  - `shmem_int_p(addr2, value2, PE 2)`
  - `shmem_int_p(addr3, value3, PE 0)`
  - `shmem_quiet()`

- **PE 1**
  - `shmem_int_p(addr4, value4, PE 0)`
  - `shmem_int_p(addr5, value5, PE 2)`

**PE K** (any PE in the system)

- Value1, value2, and value3 are delivered to target PEs and visible for PE K after the `shmem_quiet()` call.

All **Put** operations, AMOs, and store operations on symmetric data issued by a local PE to all remote PEs are guaranteed to be completed and visible once quiet returns. This operation should be used when all remote writes issued by a local PE need to be visible to all other PEs before the local PE proceeds.

### Collective synchronization over an Active set

**shmem_barrier**

**Diagram:**

- **PE 0**
  - `shmem_long_fadd(…)`
  - `shmem_int_p(…)`
  - `shmem_int_get(…)`
- **PE 1**
  - `shmem_int_p(…)`
  - `shmem_long_fadd(…)`
  - `shmem_int_get(…)`
- **PE K**
  - `shmem_long_p(…)`

**Active Set**

- All local and remote memory operations issued by PEs are guaranteed to be completed before any PE returns from the call.

All local and remote memory operations issued by all PEs within the **Active set** are guaranteed to be completed before any PE in the **Active set** returns from the call. Additionally, no PE may return from the barrier until all PEs in the **Active set** have entered the same barrier call. This operation should be used when synchronization as well as completion of all stores and remote memory updates via OpenSHMEM is required over a sub set of the executing PEs.
Collective synchronization over all PEs

`shmem_barrier_all` all

All local and remote memory operations issued by PEs are guaranteed to be completed before any PE returns from the call. Additionally no PE shall return from the barrier until all PEs have entered the same `shmem_barrier_all` call. This operation should be used when synchronization as well as completion of all stores and remote memory updates via OpenSHMEM is required over all PEs.
8.8 Distributed Locking Operations

The following section discusses OpenSHMEM locks as a mechanism to provide mutual exclusion. Three operations are available for distributed locking, `set`, `test` and `clear`.

8.8.1 SHMEM_LOCK

Releases, locks, and tests a mutual exclusion memory lock.

SYNOPSIS

C/C++:

```c
void shmem_clear_lock(long *lock);
void shmem_set_lock(long *lock);
int shmem_test_lock(long *lock);
```

FORTRAN:

```fortran
INTEGER lock, SHMEM_TEST_LOCK
CALL SHMEM_CLEAR_LOCK(lock)
CALL SHMEM_SET_LOCK(lock)
I = SHMEM_TEST_LOCK(lock)
```

DESCRIPTION

Arguments

| IN  | lock | A symmetric data object that is a scalar variable or an array of length 1. This data object must be set to 0 on all PEs prior to the first use. lock must be of type long. If you are using Fortran, it must be of default kind. |

API description

The `shmem_set_lock` routine sets a mutual exclusion lock after waiting for the lock to be freed by any other PE currently holding the lock. Waiting PEs are assured of getting the lock in a first-come, first-served manner. The `shmem_clear_lock` routine releases a lock previously set by `shmem_set_lock` after ensuring that all local and remote stores initiated in the critical region are complete. The `shmem_test_lock` function sets a mutual exclusion lock only if it is currently cleared. By using this function, a PE can avoid blocking on a set lock. If the lock is currently set, the function returns without waiting. These routines are appropriate for protecting a critical region from simultaneous update by multiple PEs.

Return Values

The `shmem_test_lock` function returns 0 if the lock was originally cleared and this call was able to set the lock. A value of 1 is returned if the lock had been set and the call returned without waiting to set the lock.

Notes

The term symmetric data object is defined in Introduction. The lock variable should always be initialized to zero and accessed only by the OpenSHMEM locking API. Changing the value of the lock variable by other means without using the OpenSHMEM API, can lead to undefined behavior.

EXAMPLES

The following example uses `shmem_lock` in a C program.

```c
#include <stdio.h>
#include <unistd.h>
```
```c
#include <shmem.h>

long L = 0;

int main(int argc, char **argv)
{
    int me, slp;
    start_pes(0);
    me = __my_pe();
    slp = 1;
    shmem_barrier_all();
    if (me == 1)
        sleep (3);
    sleep (slp);
    printf("%d: sleeping %d second%s...
", me, slp, slp == 1 ? "" : "s");
    sleep(slp);
    printf("%d: sleeping...done
", me);
    shmem_clear_lock(&L);
    shmem_barrier_all();
    return 0;
}
```

## 8.9 Deprecated API

All of these operations are deprecated and are provided for backwards compatibility. Implementations must include all items in this section and the operations should function properly, while notifying the user about deprecation of the functionality.

### 8.9.1 SHMEM_CACHE

Controls data cache utilities.

#### SYNOPSIS

**C/C++:**

```c
void shmem_clear_cache_inv(void);
void shmem_set_cache_inv(void);
void shmem_clear_cache_line_inv(void *target);
void shmem_set_cache_line_inv(void *target);
void shmem_udcflush(void);
void shmem_udcflush_line(void *target);
```

**FORTRAN:**

```fortran
CALL SHMEM_CLEAR_CACHE_INV
CALL SHMEM_SET_CACHE_INV
CALL SHMEM_SET_CACHE_LINE_INV(target)
CALL SHMEM_UDCFLUSH
CALL SHMEM_UDCFLUSH_LINE(target)
```

#### DESCRIPTION

**Arguments**

| IN | target | A data object that is local to the PE. target can be of any noncharacter type. If you are using Fortran, it can be of any kind. |

**API description**

- `shmem_set_cache_inv` enables automatic cache coherency mode.
- `shmem_set_cache_line_inv` enables automatic cache coherency mode for the cache line associated with the address of `target` only.
shmem_clear_cache_inv disables automatic cache coherency mode previously enabled by shmem_set_cache_inv or shmem_set_cache_line_inv.

shmem_udcflush makes the entire user data cache coherent.

shmem_udcflush_line makes coherent the cache line that corresponds with the address specified by target.

Return Values
None.

Notes
These routines have been retained for improved backward compatibility with legacy architectures. They are not required to be supported by implementing them as no-ops and where used, they may have no effect on cache line states.

EXAMPLES
None.
Annex A

Writing OpenSHMEM Programs

Incorporating OpenSHMEM into Programs

In this section, we describe how to write a “Hello World” OpenSHMEM program. To write a “Hello World” OpenSHMEM program we need to:

• Add the include file shmem.h (for C) or shmem.fh (for Fortran).
• Add the initialization call start_pes, (line 9) use single integer argument, 0, which is ignored.
• Use OpenSHMEM calls to query the the total number of PEs (line 10) and PE id (line 11).
• There is no explicit finalize call, either a return from main() (line 13) or an explicit exit() acts as an implicit OpenSHMEM finalization.
• In OpenSHMEM the order in which lines appear in the output is not fixed as PEs execute asynchronously in parallel.

```c
#include <stdio.h>
#include <shmem.h>      /* The shmem header file */

int main (int argc, char *argv[])
{
    int nprocs, me;
    start_pes (0);
    nprocs = shmem_n_pes ();
    me = shmem_my_pe ();
    printf ("Hello from %d of %d\n", me, nprocs);
    return 0;
}
```

Listing A.1: Expected Output (4 processors)

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

1 The unused argument is for compatibility with older SHMEM implementations.
OpenSHMEM also has a Fortran API, so for completeness we will now give the same program written in Fortran, in listing A:

```fortran
program hello
  include 'shmem.fh'
  integer :: shmem_my_pe, shmem_n_pes
  integer :: npes, me
  call start_pes (0)
  npes = shmem_n_pes ()
  me = shmem_my_pe ()
  write (*, 1000) me, npes
  1000 format ('Hello from', 1X, I4, 1X, 'of', 1X, I4)
end program hello
```

The following example shows a more complex OpenSHMEM program that illustrates the use of symmetric data objects. Note the declaration of the static short target array and its use as the remote destination in OpenSHMEM short Put. The use of the static keyword results in the target array being symmetric on PE 0 and PE 1. Each PE is able to transfer data to the target array by simply specifying the local address of the symmetric data object which is to receive the data. This aids programmability, as the address of the target need not be exchanged with the active side (PE 0) prior to the RMA (Remote Memory Access) operation. Conversely, the declaration of the short source array is asymmetric. Because the Put handles the references to the source array only on the active (local) side, the asymmetric source object is handled correctly.
```c
#include <stdio.h>
#include <shmem.h>
#define SIZE 16

int main(int argc, char* argv[])
{
    short source[SIZE];
    static short target[SIZE];
    int i, npes;
    start_pes(0);
    npes = _num_pes();
    if (_my_pe() == 0) {
        /* initialize array */
        for (i = 0; i < SIZE; i++)
            source[i] = i;
        /* static makes it symmetric */
        /* put "size" words into target on each PE */
        for (i = 1; i < npes; i++)
            shmem_short_put(target, source, SIZE, i);
    }
    else
        shmem_barrier_all(); /* sync sender and receiver */
    if (_my_pe() != 0) {
        printf("target on PE \d is \t", _my_pe());
        for (i = 0; i < SIZE; i++)
            printf("%hd \t", target[i]);
        printf("\n");
    }
    else
        shmem_barrier_all(); /* sync before exiting */
    return 0;
}
```

Listing A.3: Expected Output (4 processors)

```
target on PE 1 is 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
target on PE 2 is 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
target on PE 3 is 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```
Annex B

Compiling and Running Applications

As of now the OpenSHMEM specification is silent regarding how OpenSHMEM programs are compiled, linked and run. This section shows some examples of how wrapper programs are utilized in the OpenSHMEM Reference Implementation to compile and launch applications.

1 Compilation

Applications written in C

The OpenSHMEM Reference Implementation provides a wrapper program named oshcc, to aid in the compilation of C applications, the wrapper could be called as follows:

```
oshcc <compiler options> -o myprogram myprogram.c
```

Where the (compiler options) are options understood by the underlying C compiler.

Applications written in C++

The OpenSHMEM Reference Implementation provides a wrapper program named oshCC, to aid in the compilation of C++ applications, the wrapper could be called as follows:

```
oshCC <compiler options> -o myprogram myprogram.cpp
```

Where the (compiler options) are options understood by the underlying C++ compiler called by oshCC.

Applications written in Fortran

The OpenSHMEM Reference Implementation provides a wrapper program named oshfort, to aid in the compilation of Fortran applications, the wrapper could be called as follows:

```
oshfort <compiler options> -o myprogram myprogram.f
```

Where the (compiler options) are options understood by the underlying Fortran compiler called by oshfort.

2 Running Applications

The OpenSHMEM Reference Implementation provides a wrapper program named oshrun, to launch OpenSHMEM applications, the wrapper could be called as follows:

```
oshrun <additional options> -np <#> <program> <program arguments>
```
The program arguments for `oshrun` are:

<table>
<thead>
<tr>
<th></th>
<th>Options passed to the underlying launcher.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The number of PEs to be used in the execution.</td>
</tr>
<tr>
<td></td>
<td>The program executable to be launched.</td>
</tr>
<tr>
<td></td>
<td>Flags and other parameters to pass to the program.</td>
</tr>
</tbody>
</table>
Annex C

Undefined Behavior in OpenSHMEM

The specification provides guidelines to the expected behavior of various library routines. In cases where routines are improperly used or the input is not in accordance with the specification, undefined behavior may be observed. Depending on the implementation there are many interpretations of undefined behavior.

<table>
<thead>
<tr>
<th>Inappropriate Usage</th>
<th>Undefined Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized library</td>
<td>If OpenSHMEM is not initialized through a call to <code>start_pes</code>, subsequent accesses to OpenSHMEM routines have undefined results. An implementation may choose, for example, to try to continue or abort immediately upon the first call to an uninitialized routine.</td>
</tr>
<tr>
<td>Accessing non-existent PEs</td>
<td>If a communications routine accesses a non-existent PE then the OpenSHMEM library can choose to handle this situation in an implementation-defined way. For example, the library may issue an error message saying that the PE accessed is outside the range of accessible PEs, or may exit without a warning.</td>
</tr>
<tr>
<td>Use of non-symmetric variables</td>
<td>Some routines require remotely accessible variables to perform their function. A <code>Put</code> to a non-symmetric variable can be trapped where possible and the library can abort the program. Another implementation may choose to continue either with a warning or silently.</td>
</tr>
<tr>
<td>Non-symmetric variables</td>
<td>The symmetric memory management routines are collectives, which means that all PEs in the program must issue the same <code>shmalloc</code> call with the same size request. Program behavior after a mismatched <code>shmalloc</code> call is undefined.</td>
</tr>
</tbody>
</table>
Annex D

Interoperability with other Programming Models

1 MPI Interoperability

OpenSHMEM functions can be used in conjunction with MPI functions in the same application. For example, on SGI systems, programs that use both MPI and OpenSHMEM functions call MPI_Init and MPI_Finalize but omit the call to the start_pes function. OpenSHMEM PE numbers are equal to the MPI rank within the MPI_COMM_WORLD environment variable. Note that this precludes use of OpenSHMEM functions between processes in different MPI_COMM_WORLDS. MPI processes started using the MPI_Comm_spawn function, for example, cannot use OpenSHMEM functions to communicate with their parent MPI processes.

On SGI systems MPI jobs that use TCP/sockets for inter-host communication, OpenSHMEM functions can be used to communicate with processes running on the same host. The shmem_pe_accessible function can be used to determine if a remote PE is accessible via OpenSHMEM communication from the local PE. When running an MPI application involving multiple executable files, OpenSHMEM functions can be used to communicate with processes running from the same or different executable files, provided that the communication is limited to symmetric data objects. On these systems, static memory, such as a Fortran common block or C global variable, is symmetric between processes running from the same executable file, but is not symmetric between processes running from different executable files. Data allocated from the symmetric heap (shmalloc or shpalloc) is symmetric across the same or different executable files. The function shmem_addr_accessible can be used to determine if a local address is accessible via OpenSHMEM communication from a remote PE.

Another important feature of these systems is that the shmem_pe_accessible function returns TRUE only if the remote PE is a process running from the same executable file as the local PE, indicating that full OpenSHMEM support (static memory and symmetric heap) is available. When using OpenSHMEM functions within an MPI program, the use of MPI memory placement environment variables is required when using non-default memory placement options.
Annex E

History of OpenSHMEM

SHMEM has a long history as a parallel programming model, having been used extensively on a number of products since 1993, including Cray T3D, Cray X1E, the Cray XT3/4, SGI Origin, SGI Altix, clusters based on the Quadrics interconnect, and to a very limited extent, Infiniband based clusters.

• A SHMEM Timeline
  – Cray SHMEM
    * SHMEM first introduced by Cray Research Inc. in 1993 for Cray T3D
    * Cray is acquired by SGI in 1996
    * Cray is acquired by Tera in 2000 (MTA)
    * Platforms: Cray T3D, T3E, C90, J90, SV1, SV2, X1, X2, XE, XMT, XT
  – SGI SHMEM
    * SGI purchases Cray Research Inc. and SHMEM was integrated into SGI’s Message Passing Toolkit (MPT)
    * SGI currently owns the rights to SHMEM and OpenSHMEM
    * Platforms: Origin, Altix 4700, Altix XE, ICE, UV
    * SGI was purchased by Rackable Systems in 2009
    * SGI and Open Source Software Solutions, Inc. (OSSS) signed a SHMEM trademark licensing agreement, in 2010
  – Other Implementations
    * Quadrics (Vega UK, Ltd.)
    * Hewlett Packard
    * GPSHMEM
    * IBM
    * QLogic
    * Mellanox
    * University of Florida

• OpenSHMEM Implementations
  – SGI OpenSHMEM
  – University of Houston - OpenSHMEM Reference Implementation
  – Mellanox ScalableSHMEM
  – Portals-SHMEM
  – IBM OpenSHMEM
Annex F

Changes to this Document

1 Version 1.1

This section summarizes the changes from the OpenSHMEM specification Version 1.0 to the Version 1.1. A major change in this version is that it provides an accurate description of OpenSHMEM interfaces so that they are in agreement with the SGI specification. This version also explains OpenSHMEM’s programming, memory, and execution model. The document was throughly changed to improve the readability of specification and usability of interfaces. The code examples were added to demonstrate the usability of API. Additionally, diagrams were added to help understand the subtle semantic differences of various operations.

The following list describes the specific changes in 1.1:

- Clarifications of the completion semantics of memory synchronization interfaces.
  See Section 8.7.

- Clarification of the completion semantics of memory load and store operations in context of `shmem_barrier_all` and `shmem_barrier` routines.
  See Section 8.5 and 8.5.1.

- Clarification of the completion and ordering semantics of `shmem_quiet` and `shmem_fence`.
  See Section 8.7.1 and 8.7.

- Clarifications of the completion semantics of RMA and AMO routines.
  See Sections 8.3 and 8.4

- Clarifications of the memory model and the memory alignment requirements for symmetric data objects.
  See Section 3.

- Clarification of the execution model and the definition of a PE.
  See Section 4

- Clarifications of the semantics of `shmem_pe_accessible` and `shmem_addr_accessible`.
  See Section 8.1.3 and 8.1.4.

- Added an annex on interoperability with MPI.
  See Annex D.

- Added examples to the different interfaces.

- Clarification of the naming conventions for constant in C and Fortran.
  See Section 6 and 8.6.1.

- Added API calls: `shmem_char_p`, `shmem_char_g`.
  See Sections 8.3.1 and 8.3.4.
• Removed API calls: `shmем_char_put, shmем_char_get`.
  See Sections 8.3 and 8.3.3.

• The usage of `ptrdiff_t, size_t, and int` in the interface signature was made consistent with the description.
  See Sections 8.5, 8.3.3, and 8.3.6.

• Revised `shmем_barrier` example.
  See Section 8.5.1.

• Clarification of the initial value of `pSync` work arrays for `shmем_barrier`.
  See Section 8.5.1.

• Clarification of the expected behavior when multiple `start_pес` calls are encountered has been clarified.
  See Section 8.1.

• Corrected the definition of atomic increment operation.
  See Section 8.4.4.

• Clarification of the size of the symmetric heap and when it is set.
  See Section 8.2.

• Clarification of the integer and real sizes for `Fortran` API.
  See Sections 8.4, 8.4.1, 8.4.2, 8.4.3, 8.4.4, and 8.4.5.

• Clarification of the expected behavior on program exit.
  See Section 4, Execution Model.

• More detailed description for the progress of `OpenSHMEM` operations provided.
  See Section 4.1.

• Clarification of naming convention for non-standard interfaces and their inclusion in `shmемx.h`.
  See Section 5.

• Various fixes to `OpenSHMEM` code examples across the specification to include appropriate header files.

• Removing requirement that implementations should detect size mismatch and return error information for `shmal-loc` and ensuring consistent language.
  See Sections 8.2 and Annex C.

• `Fortran` programming fixes for examples.
  See Sections 8.5.4 and 8.6.1.

• Clarifications of the reuse `pSync` and `pWork` across collectives.
  See Sections 8.5, 8.5.2, 8.5.4 and 8.5.4.

• Name changes for UV and ICE for SGI systems.
  See Annex E.