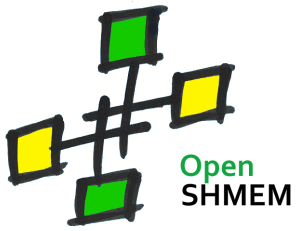
OpenSHMEM

Application Programming Interface



http://www.openshmem.org/

Version 1.0 FINAL

The comment period for Version 1.0 FINAL ends on January 31, 2012
The comment period for Version 1.1 starts on February 1, 2012

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
  http://www.defense.gov/
- Oak Ridge National Laboratory
  http://www.ornl.gov/

Authors and Collaborators

- Barbara Chapman, University of Houston
- Tony Curtis, University of Houston
- Ricardo Mauricio, University of Houston
- Swaroop Pophale, University of Houston
- Amrita Banerjee, University of Houston
- Karl Feind, SGI
- Jeff Kuehn, ORNL
- Stephen Poole, ORNL
- Lauren Smith, DoD

Acknowledgements

The OpenSHMEM development work is supported by the Oak Ridge National Laboratory Extreme Scale System Center.

The following people (listed alphabetically) have contributed ideas, criticisms and suggestions on the openshmem mailing list and in other fora:

Vikas Aggarwal; Brian W. Barrett; Christian Bell; Max Billingsley III; Mark Debbage; Mike Dubman; Dick Foster; Hal Finkel; Roger A. Golliver; Jeff Hammond; Alistair Hart; Tsai-yang Jea; Daniel Kidger; Rishi Khan; David LaFrance-Linden; John Leidel; Alexander Mikheev; Chen Qi; Duncan Roweth; Sameer Shende; Marc Snir; Lawrence Stewart; Keith D. Underwood; Brian Wibecan.

Apologies to people who have contributed but who are not acknowledged here: it is not intentional.

Contents

1 Introduction

This document defines the elements of the OpenSHMEM Application Programming Interface. The purpose of the OpenSHMEM API is to provide programmers with a standard interface for writing shared-memory parallel programs using C, C++ and Fortran.

More information about the OpenSHMEM project can be found at:
http://www.openshmem.org/

2 What is SHMEM ?

This section is an introduction to previous work on SHMEM. We begin with a quick overview of the Partitioned Global Address Space model, which is the basis for SHMEM's data sharing strategy.

2.1 Partitioned Global Address Space

Conventional Parallel Programming Models can be broadly classified into 2 types:

Shared-Memory Model: in this model all processors interact with a globally available memory space.

Distributed-Memory Model: in this model each processor has its own memory to work with and can only directly access the data that resides in its memory. When a processor needs data from another processor an explicit function call must be made to communicate with the target processor.

The current high performance computing architectures prefer a combination of the above mentioned memory models, which is referred to **Partitioned Global Address Space** or PGAS for short. In PGAS, each processing element (PE) has access to its own private local memory and also to a shared memory space. This programming model enhances performance by exposing data/thread locality. PGAS programming languages include **Unified Parallel C (UPC)**, **Co-Array Fortran (CAF)**, **Titanium**, **X-10** and **Chapel**.

More information about PGAS can be found at the PGAS Forum website.[4]

2.2 SHMEM

SHMEM stands for **SH**ared **MEM**ory. It is an API that allows the participating processes (called Processing Elements or PEs) to view a Partitioned Global Address Space. Each PE is able to see variables with a common name, but each PE has its own local copy of the variable.

The SHMEM library provides inter-processor communication using data passing and one-sided communication techniques. SHMEM differs from the Message Passing Interface (MPI), currently the most widely used communication model, in that the latter generally uses two-sided communication (MPI now also includes one-sided calls). In two-sided communication, both sides of the exchange (source and destination) are required to participate actively. The one-sided communication mechanism decouples data transfer and synchronization, reducing communication overhead, resulting in faster communication patterns. Figure 1 shows diagrams for one-sided and two-sided communications.

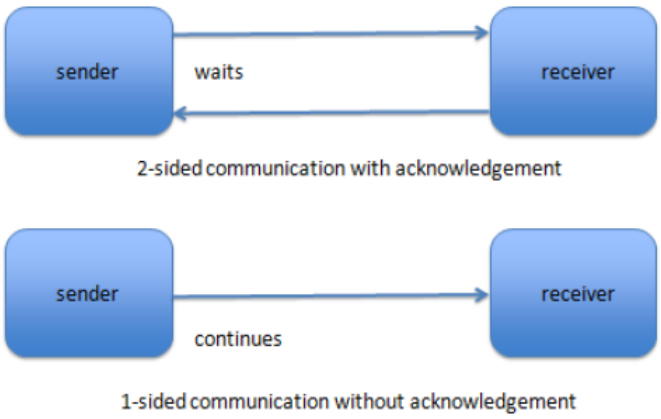


Fig. 1: Communication Scheme

The following are some of the communication operations available in SHMEM:

1. **Data Transfers**

   (a) One-sided puts : the initiator PE (active side) specifies the local data to be written to the target PE's (passive side) memory.

   (b) One-sided gets : an explicit fetch operation is used to copy a variable amount of data from a remote process and store it locally.

   Note: By avoiding the need for matching send and receive calls, SHMEM simplifies the communication process by reducing the number of calls required to have one PE interact with other PEs.

2. **Synchronization Mechanisms**

   (a) Fence: Ensures ordering of PUT operations to a specific PE.

   (b) Quiet: Ensures ordering of PUT operations to all PEs.

   (c) Barrier: A collective synchronization routine in which no PE may leave the barrier prior to all PEs entering the barrier.

3. **Collective Communication**

   (a) Broadcast: Copy a block of data from one PE to one or more target PEs.

   (b) Collection: Concatenate elements from the source array to a target array over the specified PEs.
   (c) Reduction: Perform an associative binary operation over the specified PEs.

4. **Address Manipulation**

   (a) Allocating and deallocating memory blocks in the symmetric space.

5. **Locks**

   (a) Implementation of mutual exclusion.

6. **Atomic Memory Operations**

   (a) Swap, Conditional Swap, Add and Increment

7. **Data Cache control**

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

Note: More information about OpenSHMEM routines can be found in the Library Routines section.

2.3 History of SHMEM

Cray SHMEM (MP-SHMEM, LC-SHMEM): Cray first introduced SHMEM in 1993 for its Cray T3D systems. Cray SHMEM was also used in other models: T3E, PVP and XT series.

SGI SHMEM (SGI-SHMEM): Cray Research merged with Silicon Graphics (SGI) in February 1996. At this point SHMEM was incorporated into SGI's Message Passing Toolkit (MPT). The platforms supported were - SGI Irix, Origin and Altix.

Quadrics SHMEM (Q-SHMEM): an optimized API for the Quadrics QsNet interconnect. It included SGI extensions and provided non-blocking puts and gets. A joint effort from HCS Lab & Quadrics incorporated a program profiling interface called PSHMEM that can aid in the execution analysis of SHMEM programs.

The success of SHMEM's performance attracted several vendors to provide implementations (with varying names and features) for their systems. Some of them include:

HP SHMEM: Based on the Quadrics API. It is included in the UPC product kit.

Cyclops-64 SHMEM (C64-SHMEM): this SHMEM API supports the Cyclops-64 architecture. Most of the core features of Cray SHMEM are available with some additional interfaces specific to the Cyclops-64 architecture.

IBM SHMEM: An implementation created by IBM intended for internal use only.

TurboSHMEM: This implementation uses IBM's Low-Level API (LAPI) technology to obtain optimized one-sided communication for the put/get operations. This allows applications written with the SHMEM API to run on IBM platforms with minimal source code changes.

GPSHMEM: This implementation of SHMEM aims at providing full portability of applications. It is built mostly with Cray T3D components and functionalities and provides MPI and ARMCI support. This project is no longer maintained.

3 The OpenSHMEM Project

3.1 What is OpenSHMEM ?

There are currently various SHMEM implementations available for different platforms. These versions have subtle differences from one another, and generally, code written using any one of these implementations is not directly portable to the others.

OpenSHMEM aims to address this situation by creating a process that builds a new, open specification to consolidate the various extant SHMEM versions into a widely accepted standard.

A result of this process will be an initial reference implementation based on the SGI SHMEM that can serve as a starting point for vendors and library developers. New features and enhancements can be incorporated by the community as agreed and desired. Additionally, OpenSHMEM aims to produce a portable specification enabling programmers to write SHMEM code that will run with little effort on as many different platforms as possible.

4 Language Bindings and Conformance

OpenSHMEM is available with C, C++ and Fortran bindings. The C++ interface is currently the same as that for C.

An OpenSHMEM implementation can be conformant to one or both of the interfaces. 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 Fortran and should make this clear in its documentation.

An implementation that provides both C and Fortran bindings may claim complete conformance.

5 Memory Model

The OpenSHMEM specification defines how data is stored in the memory of each PE and how data objects are made remotely accessible to all other PEs.

Data objects can be stored in a private local memory address or in a remotely accessible memory address space. Objects in the private address space can only be accessed by the PE itself; these data objects cannot be accessed by other PEs via OpenSHMEM routines. Remotely accessible objects, however, can be accessed by remote PEs using OpenSHMEM routines. Remotely accessible data objects are also known as Symmetric Objects. An object is symmetric if it has a corresponding object with the same type, size and offset on all other PEs. Examples of Symmetric Objects are static and global variables in C and C++, which are often allocated at the same address on all PEs where the program is being executed (*e.g.* in the ELF executable format). See Figure 2 for an example of how Symmetric Memory Objects may be arranged in memory.

OpenSHMEM routines allow the creation of dynamically allocated Symmetric data objects. These objects are created in a special memory region called the Symmetric Heap, which is created during execution at locations determined by the implementation, meaning the Symmetric Heap may be in different memory regions on different PEs. OpenSHMEM has nothing to say regarding the underlying memory layout; it is up to the implementation to decide how to handle the Symmetric Heap.

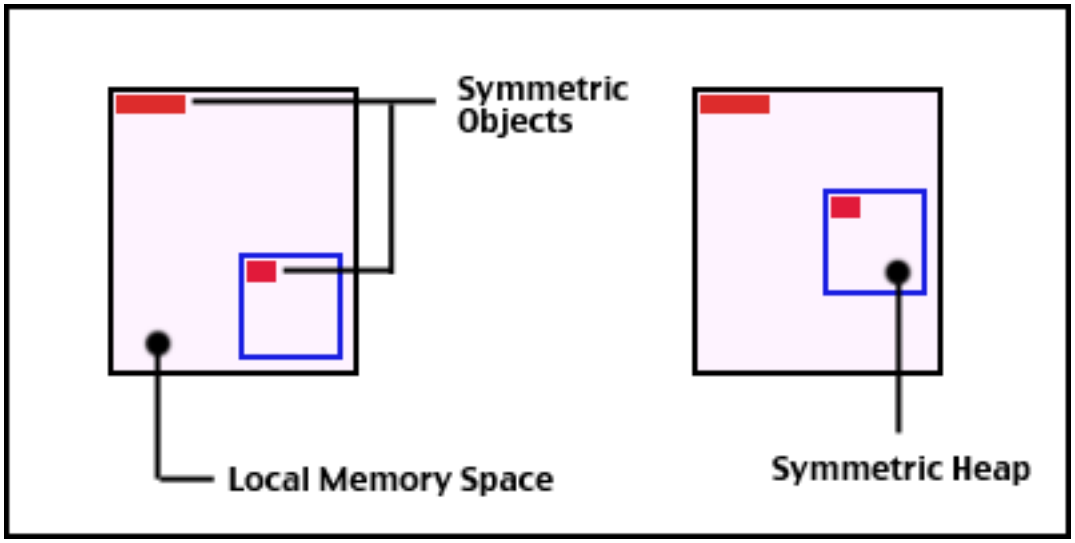


Fig. 2: Example of Symmetric Objects

6 Execution Model

This section describes the Execution Model of an OpenSHMEM application.

OpenSHMEM uses a Single Process Multiple Data (SPMD) approach to express parallelism. An OpenSHMEM application makes use of multiple processors, referred to as Processing Elements or PEs, to complete operations in parallel.

OpenSHMEM requires initialization before using any of the library routines. To this end, the program issues a call to the **start_pes()** routine. **start_pes()** performs any required initialization steps, such as setting up the symmetric heap for every PE and creating the PE numbers. The symmetric heap is one of the memory spaces that is remotely accessible by all PEs. The symmetric heap is discussed further in the Memory Model section. The PE numbers are the identifiers used to refer to each of the PEs involved in the execution. These PE numbers are integers assigned in a monotonically increasing manner from zero to the total number of PEs minus 1.

Data transfer in OpenSHMEM is possible through several one-sided put (for write) and get (for read) operations, as well as various collective routines such as broadcasts and reductions.

Query routines are available to gather information about the execution. OpenSHMEM also provides synchronization routines to coordinate data transfers and other operations.

It is up to the implementation how to handle the finalization of the OpenSHMEM library and any other resources initialized by the library: there is currently no explicit call for the programmer.

6.1 Communication Progress

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 long computation with no OpenSHMEM calls. Other PEs must be able to communicate (put/get, collective, atomic) with that computationally-bound PE without that PE issuing any explicit OpenSHMEM calls.

OpenSHMEM communication calls involving that PE must progress regardless of when that PE next engages in an OpenSHMEM call.

Note to implementers: progress will often be ensured through the use of a dedicated progress thread in software, or through network hardware that offloads communication handling from processors.

6.2 Atomicity Guarantees

OpenSHMEM contains a number of routines that operate on symmetric data atomically. 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's or outside.

For example: during the execution of a 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 1 on the target PE, at which point other atomic operations may then modify that "x".

7 Undefined Behavior

7.1 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.

Inappropriate Usage	**Undefined Behavior**
Uninitialized library	If OpenSHMEM is not initialized through a call to **start_pes()**, 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.
Accessing non-existent PEs	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.
Use of non-symmetric variables	Some routines require remotely accessible variables to perform their function. A “put” 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.
Non-symmetric Variables	The symmetric memory management routines are collectives, which means that all PEs in the program must issue the same shmalloc() call with the same size request. OpenSHMEM implementations should detect the size mismatch and return error information to the caller. Implementations may also produce an error message. Program behavior after a mismatched shmalloc() call is undefined.

8 Library Routines

8.1 Initialization Routines

8.1.1 start_pes

Summary

Initializes OpenSHMEM.

Synopsis

C/C++:

```
void start_pes(int npes);
```

Fortran:

```
INTEGER npes

CALL START_PES(npes)
```

Parameters

npes Unused. Should be set to 0.

Constraints

- If **start_pes()** is called multiple times, subsequent calls have no effect.
- An OpenSHMEM application must make a call to **start_pes()** before being able to call any other OpenSHMEM routine. Calling another OpenSHMEM library routine before calling **start_pes()** results in undefined behavior.

Effect

Initializes the execution environment of the PE. This routine is responsible *inter alia* for setting up the symmetric heap on the calling PE, and the creation of the PE numbers. Upon successful return from this routine, the calling PE will be able to communicate with and transfer data to other PEs.

Return Values

None

8.2 Query Routines

The OpenSHMEM query routines provide information about the program execution.

8.2.1 num_pes

Summary

Returns the number of processing elements (PEs) used to run the application.

Synopsis

C/C++:

```
int _num_pes(void);
```

Fortran:

```
INTEGER I

I = NUM_PES()
```

Parameters

None.

Constraints

None.

Effect

num_pes() returns the total number of PEs running in an application.

Return Values

The total number of PEs running the application. PEs are numbered 0..(n-1).

8.2.2 my_pe

Summary

Returns the PE number of the calling PE.

Synopsis

C/C++:

```
int _my_pe(void);
```

Fortran:

```
INTEGER I

I = MY_PE()
```

Parameters

None.

Constraints

None.

Effect

my_pe() returns the PE number of the calling PE. PEs are numbered from 0..(n-1).

Return Values

The PE number of the calling PE.

8.3 Accessibility Query Routines

8.3.1 shmem_pe_accessible

Summary

This routine determines if a remote PE is reachable from the calling PE.

Synopsis

C/C++:

```
int shmem_pe_accessible(int pe);
```

Fortran:

```
LOGICAL LOG, SHMEM_PE_ACCESSIBLE
INTEGER pe

LOG = SHMEM_PE_ACCESSIBLE(pe)
```

Parameters

pe PE number of the remote PE.

Constraints

- **pe** must be a PE number. For more information about how PE numbers are assigned please refer to the Execution Model section.

Effect

This routine returns a value that indicates whether the calling PE is able to perform OpenSHMEM communication operations with the remote PE.

Return Values

In C/C++, **shmem_pe_accessible()** returns 1 if the specified PE is a valid remote PE for OpenSHMEM functions; otherwise, it returns 0. In Fortran, **shmem_pe_accessible()** returns **.TRUE.** if the specified PE is a valid remote PE for OpenSHMEM functions; otherwise, it returns **.FALSE.**.

8.3.2 shmem_addr_accessible

Summary

This routine indicates if an address is accessible via OpenSHMEM operations from the specified remote PE.

Synopsis

C/C++:

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

Fortran:

```
LOGICAL LOG, SHMEM_ADDR_ACCESSIBLE
INTEGER pe

LOG = SHMEM_ADDR_ACCESSIBLE(addr, pe)
```

Parameters

addr The address to the memory block in the local symmetric heap.

pe PE number of the remote PE.

Constraints

- **pe** must be a PE number. For more information about how PE numbers are assigned please refer to the Execution Model section.

Effect

shmem_addr_accessible() determines if the remote PE **pe** is accessible via OpenSHMEM communication routines and that the address **addr** is in a symmetric segment with respect to the remote PE **pe**.

Return Values

In C/C++, **shmem_addr_accessible()** returns 1 if the data object at address **addr** is symmetric and can be accessed via OpenSHMEM functions; otherwise, it returns 0. In Fortran, **shmem_addr_accessible() .TRUE.** if the data object at address **addr** is symmetric and can be accessed via OpenSHMEM functions; otherwise, it returns **.FALSE.**.

8.4 Symmetric Heap Routines

The OpenSHMEM Symmetric Heap routines manage memory blocks inside the symmetric heap. The total size of the symmetric heap is determined at program start.

8.4.1 shmalloc

Summary

This routine allocates a block of memory in the symmetric heap of the calling PE.

Synopsis

C/C++:

```
void *shmalloc(size_t size);
```

Parameters

size Size of the requested memory block, in bytes.

Constraints

- All PEs must call this routine at the same point of the execution path; otherwise, undefined behavior results.
- All PEs must call this routine with the same **size** value; otherwise, undefined behavior results.
- The parameter **size** must be less than or equal to the amount of symmetric heap space available for the calling PE; otherwise shmalloc returns NULL.

Effect

The **shmalloc()** routine allocates a block of memory of at least **size** bytes from the symmetric heap of the calling PE, and returns a pointer to the allocated block.

shmalloc() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric allocation, and that the remote memory on all the PEs is available to the other PEs.

Return Values

The **shmalloc()** routine returns a pointer to the allocated block; otherwise a null pointer is returned if no block could be allocated.

8.4.2 shmemalign

Summary

This routine allocates a block from the symmetric heap with a byte alignment specified by the programmer.

Synopsis

C/C++:

```
void *shmemalign(size_t alignment, size_t size);
```

Parameters

alignment Size of the alignment block, in bytes.

size Size for the memory block, in bytes.

Constraints

- Values for parameters **alignment** and **size** must be positive integer values.
- **alignment** is a power of 2; $power \geq 3$.
- The parameter **size** must be less than or equal to the amount of symmetric heap space available for the calling PE; otherwise shmemalign returns NULL.

Effect

The **shmemalign()** routine allocates a memory block of **size** bytes in the symmetric heap, with an alignment of **alignment** bytes.

shmemalign() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric allocation, and that the remote memory on all the PEs is available to the other PEs.

Return Values

This routine returns a pointer to the allocated block of memory; otherwise a null pointer is returned.

8.4.3 shrealloc

Summary

This routine expands or reduces the size of the block to which **ptr** points, depending on the provided **size** parameter.

Synopsis

C/C++:

```
void *shrealloc(void *ptr, size_t size);
```

Parameters

ptr Pointer to a memory block previously allocated with **shmalloc()** or **shrealloc()** to be reallocated.

size New size for the memory block, in bytes.

Constraints

- All PEs must call this routine at the same point of the execution path; otherwise, undefined behavior results.
- All PEs must call this routine with the same parameters; otherwise different symmetric heap addresses may be returned to each PE.
- The parameter **size** must be an integer greater than or equal to zero.
- The parameter **size** must be lesser than or equal to the amount of symmetric heap space available for the calling PE; otherwise shrealloc returns NULL.

Effect

The **shrealloc()** routine changes the size of the memory block to which **ptr** points to, and returns a pointer to the memory block. The contents of the memory block are preserved 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. In case **shrealloc()** is unable to extend the size of the memory block at its current location, the routine may move the block elsewhere in the symmetric heap while ensuring that contents of the block are preserved. In case of relocation, **shrealloc()** will return a pointer to the new location.

If **ptr** is null, **shrealloc()** behaves exactly like **shmalloc()**. If **size** is 0 and **ptr** is not null, the memory block is deallocated. Otherwise, if **ptr** does not match a pointer earlier returned by a symmetric heap function, or if the space has already been deallocated, **shrealloc()** will return a null pointer. If the space cannot be allocated, the block to which **ptr** points to is unchanged.

shrealloc() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric allocation, and that the remote memory on all the PEs is available to the other PEs.

8.4.4 shfree

Summary

Frees a memory block previously allocated in the symmetric heap.

Synopsis

C/C++:

```
void shfree(void *ptr);
```

Parameters

ptr Pointer to a memory block previously allocated with **shmalloc()** or **shrealloc()** to be deallocated.

Constraints

None.

Effect

This routine causes the block to which **ptr** points to, to be deallocated; that is, made available for further allocation.

If **ptr** is a null pointer, no action occurs; otherwise, if the argument does not match a pointer earlier returned by a symmetric heap function, or if the space has already been deallocated, **shfree()** returns.

shfree() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric deallocation within the heap.

Return Values

None.

8.4.5 SHPALLOC

Summary

This routine allocates a block of memory in the symmetric heap of the calling PE.

Synopsis

Fortran:

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

CALL SHPALLOC(addr, length, errcode, abort)
```

Parameters

addr First word address of the allocated block (output variable).

length Number of words of memory requested (input variable). One word is 32 bits.

errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output variable).

abort Abort code; nonzero requests abort on error; 0 requests an error code (input variable).

Constraints

- All PEs must call this routine at the same point of the execution path; otherwise, undefined behavior results.
- All PEs must call this routine with the same parameters; otherwise different symmetric heap addresses may be returned to each PE..
- **length** must be lesser than or equal to the amount of symmetric heap space available for the calling PE.

Effect

The **SHPALLOC()** routine allocates a block of memory in the symmetric heap of the calling PE, and returns the address of the allocated block via the **addr** parameter.

shpalloc() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric allocation, and that the remote memory on all the PEs is available to the other PEs.

Return Values

The **SHPALLOC()** routine returns the address of the allocated memory block via the **addr** parameter and a result code via the **errcode** parameter. The possible values for errcode are:

0 Operation was successful .

-1 Length is not an integer greater than 0.

-2 No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).

8.4.6 SHPDEALLC

Summary

Frees a memory block previously allocated in the symmetric heap.

Synopsis

Fortran:

```
POINTER (addr, A(1))
INTEGER errcode, abort

CALL SHPDEALLC(addr, errcode, abort)
```

Parameters

addr First word address of the block to deallocate (input).

errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).

abort Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Constraints

- All PEs must call this routine at the same point of the execution path; otherwise, undefined behavior results.
- All PEs must call this routine with the same parameters; otherwise different symmetric heap addresses may be returned to each PE.
- The parameter **addr** must be the address of a block of memory allocated in the symmetric heap.
- To maintain symmetric heap consistency, all processing elements (PEs) in a program must call **SHPDEALLC()** with the same value of **addr**; if any PEs fail to call this routine, it may result in undefined behavior.

Effect

This routine causes the block at address **addr**, to be deallocated, that is, made available for further allocation. If **addr** is not an address in the symmetric heap, an error code is returned via the parameter **errcode**.

shdeallc() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric deallocation.

Return Values

This routine returns a result code via the **errcode** parameter. Possible values for errcode are:

0 Operation was successful .

-3 Address is outside the bounds of the symmetric heap.

-4 Block is already free.

-5 Address is not at the beginning of the block.

8.4.7 SHPCLMOVE

Summary

This routine expands or reduces the size of the memory block with address **addr**, depending on the provided **length** parameter.

Synopsis

Fortran:

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

CALL SHPCLMOVE(addr, length, status, abort)
```

Parameters

addr On entry, first word address of the block to change; on exit, the new address of the block if it was moved. (input and output)

length Requested new total length in words (input). One word is 32 bits.

status Status code. See Return Values below for possible status codes (output).

abort Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Constraints

- All PEs must call this routine at the same point of the execution path; otherwise, undefined behavior results.
- All PEs must call this routine with the same parameters; otherwise different symmetric heap addresses may be returned to each PE.
- The parameter **addr** must be the address of a block of memory allocated in the symmetric heap.

Effect

The **SHPCLMOVE()** routine 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. The function may move the memory block to a new location, in which case the address of the new location is returned.

shpclmove() calls **shmem_barrier_all()** before returning to ensure that all the PEs participate. This guarantees symmetric allocation, and that the remote memory on all the PEs is available to the other PEs.

Return Values

The address of the new block passed back via the parameter **addr**.

Possible codes returned via the **status** parameter:

0 The memory block was resized at its initial location in the symmetric heap.

1 The memory block was moved to a new location in the symmetric heap.

-1 Length is not an integer greater than 0.

-2 No more memory is available from the system (checked if the block cannot be extended and the free space list does not include a large enough block).

-3 Address is outside the bounds of the symmetric heap.

-4 Block is already free.

-5 Address is not at the beginning of a block.

8.5 Remote Pointer Operations

8.5.1 shmem_ptr

Summary

Returns a pointer to a data object of a remote PE.

Synopsis

C/C++:

```
void *shmem_ptr(void *target, int pe);
```

Fortran:

```
POINTER (PTR, POINTEE)
INTEGER pe

PTR = SHMEM_PTR(target, pe)
```

Parameters

target Address of the symmetric data object.

pe The PE number of the remote PE.

Constraints

- The **shmem_ptr()** function only returns a non-NULL value on systems where ordinary memory loads and stores are used to implement OpenSHMEM put and get operations.
- **target** must be the address of a symmetric data object.
- **pe** must be a PE number. For more information about how PE numbers are assigned please refer to the Execution Model section.

Effect

The **shmem_ptr()** routine returns an address that can be used to directly reference **target** on the remote PE **pe**. The programmer must be able to assign this address to a pointer and perform ordinary loads and stores to this remote address.

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 a 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

Returns a pointer to the data object on the remote PE or NULL if the remote object cannot be accessed directly.

8.6 Elemental Put Routines

8.6.1 shmem_TYPE_p

Summary These routines provide a low latency mechanism to write basic types (*short*, *int*, *float*, *double*, *long*, *long long*, *long double*) to symmetric data objects on remote PEs.

Synopsis C/C++:

```
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_float_p(float *addr, float value, int pe);
void shmem_double_p(double *addr, double value, int pe);
void shmem_longlong_p(long long *addr, long long value, int pe);
void shmem_longdouble_p(long double *addr, long double value, int pe);
```

Parameters

addr Address of the symmetric data object where to save the data on the remote **pe**.

value The value to be transferred to **addr** on the remote **pe**.

pe PE number of the remote PE.

Constraints

- **addr** must be the address of a symmetric data object.
- **pe** must be a PE number. For more information about how PE numbers are assigned please refer to the Execution Model section.

Effect The **shmem_TYPE_p()** routines write **value** to a symmetric array element or scalar data object of the remote PE indicated by the parameter **pe**. These routines start the remote transfer and may return before the data is delivered to the remote PE.

Return Values None

8.7 Block Data Put Routines

8.7.1 shmem_put

Summary

These routines copy contiguous data from a local object to an object on the destination PE.

Synopsis

C/C++:

```
void shmem_char_put(char *target, const char *source, size_t nelems, int
   pe);
void shmem_short_put(short *target, const short *source, size_t nelems, int
   pe);
void shmem_int_put(int *target, const int *source, size_t nelems, int pe);
void shmem_long_put(long *target, const long *source, size_t nelems, int
   pe);
void shmem_float_put(float *target, const float *source, size_t nelems, int
   pe);
void shmem_double_put(double *target, const double *source, size_t nelems,
   int pe);
void shmem_longlong_put(long long *target, const long long *source, size_t
   nelems, int pe);
void shmem_longdouble_put(long double *target, const long double *source,
   size_t nelems, int pe);
```

```
void shmem_put32(void *target, const void *source, size_t nelems, int pe);
void shmem_put64(void *target, const void *source, size_t nelems, int pe);
void shmem_put128(void *target, const void *source, size_t nelems, int pe);
void shmem_putmem(void *target, const void *source, size_t nelems, int pe);
```

Fortran:

```
INTEGER nelems, pe

CALL SHMEM_CHARACTER_PUT(target, source, nelems, pe)
CALL SHMEM_COMPLEX_PUT(target, source, nelems, pe)
CALL SHMEM_DOUBLE_PUT(target, source, nelems, pe)
CALL SHMEM_INTEGER_PUT(target, source, nelems, pe)
CALL SHMEM_LOGICAL_PUT(target, source, nelems, pe)
CALL SHMEM_REAL_PUT(target, source, nelems, pe)
CALL SHMEM_PUT(target, source, nelems, pe)
CALL SHMEM_PUT4(target, source, nelems, pe)
CALL SHMEM_PUT8(target, source, nelems, pe)
CALL SHMEM_PUT32(target, source, nelems, pe)
CALL SHMEM_PUT64(target, source, nelems, pe)
CALL SHMEM_PUT128(target, source, nelems, pe)
CALL SHMEM_PUTMEM(target, source, nelems, pe)
```

Parameters

target Address of the symmetric data object where to save the data on the remote **pe**.

source Address of the data to be transferred to the remote data object.

nelems The number of elements in the **target** and **source** objects.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.
- **source** must have the same type as **target**.
- **nelems** must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.
- **pe** must be a PE number. For more information about how PE numbers are assigned please refer to the Execution Model section.

Effect

These routines transfer **nelems** elements of the data object at address **source** on the calling PE, to the data object at address **target** on the remote PE **pe**. These routines start the remote transfer and may return before the data is delivered to the remote PE. The delivery of data into the data object on the destination PE from different put calls may occur in any order. Because of this, two successive put operations may deliver data out of order unless a call to **shmem_fence()** is introduced between the two calls.

Return Values

None

8.8 Strided Put Routines

8.8.1 shmem_iput

Summary

These routines copy strided data from the local PE to a strided data object on the destination PE.

Synopsis

C/C++:

```
void shmem_short_iput(short *target, const short *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_float_iput(float *target, const float *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_double_iput(double *target, const 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_longdouble_iput(long double *target, const long double *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);
```

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_LOGICAL_IPUT(target, source,  tst,  sst, nelems, pe)
CALL SHMEM_REAL_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)
```

Parameters

target Address of the symmetric data object where to save the data on the remote **pe**.

source Address of the data to be transferred to the remote data object.

tst The stride between consecutive elements in the **target** array. **tst** must be of type integer. If you are using Fortran, it must be a default integer value.

sst The stride between consecutive elements in the **source** array. **sst** must be of type integer. If you are using Fortran, it must be a default integer value.

nelems Number of elements in the **target** and **source** objects. **nelems** must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.

- **source** must have the same type as **target**.
- The strides **tst** and **sst** are scaled by the element size of the **target** and **source** arrays respectably. A value of 1 indicates contiguous data. **tst** and **sst** must be of type integer. If you are using Fortran, it must be a default integer value.
- **pe** must be a PE number. For more information about how PE numbers are assigned please refer to the Execution Model section.
- Depending on the routine being called, **source** and **target** must conform to the following typing constraints:
  - In **shmem_iput32()** and **shmem_iput4()** they can only have a non character type that has a storage size equal to 32 bits.
  - In **shmem_iput64()** and **shmem_iput8()** they can only have a non character type that has a storage size equal to 64 bits.
  - In **shmem_iput128()** they can only have a non character type that has a storage size equal to 128 bits.
  - In **shmem_short_iput()** they must be of type short.
  - In **shmem_int_iput()** they must be of type int.
  - In **shmem_float_iput()** they must be of type float.
  - In **shmem_double_iput()** they must be of type double.
  - In **shmem_long_iput()** they must be of type long.
  - In **shmem_longlong_iput()** they must be of type long long.
  - In **shmem_longdouble_iput()** they must be of type long double.
  - In **SHMEM_COMPLEX_IPUT()** they must be of type complex of default size.
  - In **SHMEM_DOUBLE_IPUT()** they must be of type double precision.
  - In **SHMEM_INTEGER_IPUT()** they must be of type integer.
  - In **SHMEM_LOGICAL_IPUT()** they must be of type logical.
  - In **SHMEM_REAL_IPUT()** they must be of type real.

Effect

The **shmem_iput()** routines read the elements of a local array (**source**) and write them to a remote array (**target**) on the PE indicated by **pe**. These 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.

Return Values

None.

8.9 Elemental Data Get Routines

8.9.1 shmem_TYPE_g

Summary

These routines provide a low latency mechanism to retrieve basic types (*short*, *int*, *float*, *double*, *long*) from symmetric data objects on remote PEs.

Synopsis

C/C++:

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

Parameters

addr Address of the symmetric data object that contains the data to be read.

pe PE number of the remote PE.

Constraints

- **addr** must be the address of a symmetric data object.

Effect

Retrieves the value at the symmetric address **addr** of the remote PE **pe**.

Return Value

The value at the symmetric address **addr** on PE **pe.**

8.10 Block Data Get Routines

8.10.1 shmem_get

Summary

These routines retrieve data from a contiguous data object on a remote PE.

Synopsis

C/C++:

```
void shmem_char_get(char *target, const char *source, size_t nelems, int
   pe);
void shmem_short_get(short *target, const short *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_float_get(float *target, const float *source, size_t nelems, int
   pe);
void shmem_double_get(double *target, const 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_longdouble_get(long double *target, const long double *source,
   size_t nelems, int pe);
void shmem_get32(void *target, const void *source, size_t nelems, int pe);
void shmem_get64(void *target, const void *source, size_t nelems, int pe);
void shmem_get128(void *target, const void *source, size_t nelems, int pe);
void shmem_getmem(void *target, const void *source, size_t nelems, int pe);
```

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_INTEGER_GET(target, source, nelems, pe)
CALL SHMEM_GET4(target, source, nelems, pe)
CALL SHMEM_GET8(target, source, nelems, pe)
CALL SHMEM_GET32(target, source, nelems, pe)
CALL SHMEM_GET64(target, source, nelems, pe)
CALL SHMEM_GET128(target, source, nelems, pe)
CALL SHMEM_GETMEM(target, source, nelems, pe)
CALL SHMEM_LOGICAL_GET(target, source, nelems, pe)
CALL SHMEM_REAL_GET(target, source, nelems, pe)
```

Parameters

target Address of the local data object in which to save the data.

source Address of the symmetric data object on the remote **pe** with the data to be retrieved.

nelems Number of elements in the **target** and **source** arrays.

pe Identifier of the remote PE.

Constraints

- **source** must be the address of a symmetric data object.
- In C/C++ **nelems** must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.

Effect

The **shmem_get()** routines transfer **nelems** elements of the data object at address **source** on the remote PE (**pe**), to the data object at address **target** on the local PE. These routines return after the data has been copied to address **target** on the local **pe**.

Return Values

After successful completion, the retrieved data will be available at address **target**.

8.11 Strided Get Routines

8.11.1 shmem_iget

Summary

The strided get routines copy strided data located on a remote PE to a local strided data object.

Synopsis

C/C++:

```
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_short_iget(short *target, const short *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_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_longlong_iget(long long *target, const long 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);
```

Fortran:

```
INTEGER tst, sst, nelems, pe

CALL SHMEM_IGET4(target, source, tst, sst, nelems, pe)
CALL SHMEM_IGET8(target, source, tst, sst, nelems, pe)
CALL SHMEM_IGET32(target, source, tst, sst, nelems, pe)
CALL SHMEM_IGET64(target, source, tst, sst, nelems, pe)
CALL SHMEM_IGET128(target, source, 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)
```

Parameters

target Address of the data object in which to save the data on the local **pe**.

source Address of the symmetric data object on the remote **pe** with the data to be retrieved.

tst The stride between consecutive elements in the **target** array.

sst The stride between consecutive elements in the **source** array.

nelems Number of elements in the **target** and **source** objects.

pe PE number of the remote PE.

Constraints

- **source** must be the address of a symmetric data object.
- **source** must have the same type as **target**.
- In C/C++ **tst** and **sst** must be of type integer. If you are using Fortran, they must be a default integer value.
- The strides **tst** and **sst** are scaled by the element size of the **target** and **source** arrays respectably. A value of 1 indicates contiguous data. **tst** and **sst** must be of type integer. If you are using Fortran, it must be a default integer value.
- In C/C++ **nelems** must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.
- Depending on the routine being called, **source** and **target** must conform to the following typing constraints:
  - In **shmem_iget32()** and **shmem_iget4()** they can only have a non character type that has a storage size equal to 32 bits.
  - In **shmem_iget64()** and **shmem_iget8()** they can only have a non character type that has a storage size equal to 64 bits.

- In **shmem_iget128()** they can only have a non character type that has a storage size equal to 128 bits.
- In **shmem_short_iget()** they must be of type short.
- In **shmem_int_iget()** they must be of type int.
- In **shmem_float_iget()** they must be of type float.
- In **shmem_double_iget()** they must be of type double.
- In **shmem_long_iget()** they must be of type long.
- In **shmem_longlong_iget()** they must be of type long long.
- In **shmem_longdouble_iget()** they must be of type long double.
- In **SHMEM_COMPLEX_IGET()** they must be of type complex of default size.
- In **SHMEM_DOUBLE_IGET()** they must be of type double precision.
- In **SHMEM_INTEGER_IGET()** they must be of type integer.
- In **SHMEM_LOGICAL_IGET()** they must be of type logical.
- In **SHMEM_REAL_IGET()** they must be of type real.

Effect

The strided get routines retrieve array data available at address **source** on remote PE (**pe**). The elements of the **source** array are separated by a stride **sst**. Once the data is received, it is stored at the local memory address **target**, separated by stride **tst**. The routines return when the data has been copied into the local **target** array.

Return Values

Upon return of this routine, the data object at address **target** will contain the data retrieved from the remote memory address **source**.

8.12 Atomic Memory fetch-and-operate Routines

This section describes the OpenSHMEM Atomic fetch-op Routines. These routines allow operations on a symmetric object guaranteeing that another process will not update **target** between the time of the fetch and the update.

8.12.1 shmem_swap

Summary

Performs an atomic swap operation.

Synopsis

C/C++:

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

Fortran:

```
INTEGER pe

INTEGER SHMEM_SWAP
ires = SHMEM_SWAP(target, value, pe)

INTEGER(KIND=4) SHMEM_INT4_SWAP
ires = SHMEM_INT4_SWAP(target, value, pe)

INTEGER(KIND=8) SHMEM_INT8_SWAP
ires = SHMEM_INT8_SWAP(target, value, pe)

REAL(KIND=4) SHMEM_REAL4_SWAP
res = SHMEM_REAL4_SWAP(target, value, pe)

REAL(KIND=8) SHMEM_REAL8_SWAP
res = SHMEM_REAL8_SWAP(target, value, pe)
```

Parameters

target Address of the symmetric data object to be updated on the remote **pe**.

value Value to be atomically written to the remote PE.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.

- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the target data types are as follows:
  - For **SHMEM_INT4_SWAP() target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_SWAP() target** must be of type Integer, with element size of 8 bytes.
  - For **SHMEM_REAL4_SWAP() target** must be of type Real, with element size of 4 bytes.
  - For **SHMEM_REAL8_SWAP() target** must be of type Real, with element size of 8 bytes.
- **value** must be the same type as **target**.
- This process must be carried out guaranteeing that it will not be interrupted by any other operation.

Effect

The atomic swap routines write **value** to address **target** on PE **pe**, and return the previous contents of **target.** The operation must be completed without the possibility of another process updating **target** between the time of the fetch and the update.

Return Values

Returns the previous value of **target**.

8.12.2 shmem_cswap

Summary

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

Synopsis

C/C++:

```
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(longlong *target, longlong cond, longlong value,
   int pe);
```

Fortran:

```
INTEGER pe

INTEGER(KIND=4) SHMEM_INT4_CSWAP
ires = SHMEM_INT4_CSWAP(target, cond, value, pe)

INTEGER(KIND=8) SHMEM_INT8_CSWAP
ires = SHMEM_INT8_CSWAP(target, cond, value, pe)
```

Parameters

target Address of the symmetric data object to be updated on the remote **pe**.

cond **cond** is compared to the remote target value.

value The value to be atomically written to the remote PE.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the target data types are as follows:
  - For **SHMEM_INT4_CSWAP() target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_CSWAP() target** must be of type Integer, with element size of 8 bytes.
- **value** and **cond** must be the same type as **target**.
- This process must be carried out guaranteeing that it will not be interrupted by any other operation.

Effect

The conditional swap routines write **value** to address **target** on PE **pe**, and return the previous contents of **target.** The replacement must occur only if **cond** is equal to **target**; otherwise **target** is left unchanged. In either case, the routine must return the initial value of **target**. The operation must be completed without the possibility of another process updating **target** between the time of the fetch and the update.

Return Values

Returns the initial value of **target.**

8.12.3 shmem_fadd

Summary

These routines perform an atomic fetch-and-add operation.

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(KIND=4) SHMEM_INT4_FADD
ires = SHMEM_INT4_FADD(target, value, pe)

INTEGER(KIND=8) SHMEM_INT8_FADD
ires = SHMEM_INT8_FADD(target, value, pe)
```

Parameters

target Address of the symmetric data object where to save the data on the remote **pe**.

value The value to be atomically added to the value at address **target**.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the target data types are as follows:
  - For **SHMEM_INT4_FADD() target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_FADD() target** must be of type Integer, with element size of 8 bytes.
- **value** must be the same type as **target**.
- This process must be carried out guaranteeing that it will not be interrupted by any other operation.

Effect

The fetch and add routines retrieve the value at address **target** on PE **pe**, and update **target** with the result of adding **value** to the retrieved value. The operation must be completed without the possibility of another process updating **target** between the time of the fetch and the update.

Return Values

Returns the initial value of **target.**

8.12.4 shmem_finc

Summary

These routines perform a fetch-and-increment operation.

Synopsis

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:

```
INTEGER pe

INTEGER(KIND=4) SHMEM_INT4_FINC
ires = SHMEM_INT4_FINC(target4, pe)

INTEGER(KIND=8) SHMEM_INT8_FINC
ires = SHMEM_INT8_FINC(target8, pe)
```

Parameters

target Address of the symmetric data object where to save the data on the remote **pe**.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the target data types are as follows:
  - For **SHMEM_INT4_FINC() target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_FINC() target** must be of type Integer, with element size of 8 bytes.
- This process must be carried out guaranteeing that it will not be interrupted by any other operation.

Effect

The fetch and increment routines retrieve the value at address **target** on PE **pe**, and update **target** with the result of incrementing the retrieved value by one. The operation must be completed without the possibility of another process updating **target** between the time of the fetch and the update.

Return Values

Returns the initial value of **target.**

8.13 Atomic Memory Operation Routines

8.13.1 shmem_add

Summary

These routines perform an atomic add operation.

Synopsis

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:

```
INTEGER pe

SHMEM_INT4_ADD(target, value, pe)
SHMEM_INT8_ADD(target, value, pe)
```

Parameters

target Address of the symmetric data object where to save the data on the remote **pe**.

value The value to be atomically added to target.

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.

Constraints

- **target** must be the address of a symmetric data object.
- If using C/C++, the type of **var** must match that implied in the Synopsis section. When calling from Fortran, the data type of **var** must be as follows:
  - For **SHMEM_INT4_ADD()**, **var** must be of type Integer, with element size of 4 bytes.

  - For **SHMEM_INT8_ADD()**, **var** must be of type Integer, with element size of 8 bytes.

- **value** must be the same type as **target**.
- This process must be carried out guaranteeing that it will not be interrupted by any other operation.

Effect

The atomic add routines add **value** to the data at address **target** on PE **pe.** The operation must be completed without the possibility of another process updating **target** between the time of the fetch and the update.

Return Values

None.

8.13.2 shmem_inc

Summary

These routines perform an atomic increment operation on a remote data object.

Synopsis

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:

```
INTEGER pe

INTEGER(KIND=4) target4
CALL SHMEM_INT4_INC(target4, pe)

INTEGER(KIND=8) target8
CALL SHMEM_INT8_INC(target8, pe)
```

Parameters

target Address of the symmetric data object where to save the data on the remote **pe**.

pe PE number of the remote PE.

Constraints

- **target** must be the address of a symmetric data object.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the target data types are as follows:
  - For **SHMEM_INT4_INC() target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_INC() target** must be of type Integer, with element size of 8 bytes.
- This process must be carried out guaranteeing that it will not be interrupted by any other operation.

Effect

The atomic increment routines replace the value of **target** with its value incremented by one. The operation must be completed without the possibility of another process updating **target** between the time of the fetch and the update.

Return Values

None.

8.14 Point-to-Point Synchronization Routines

The point-to-point synchronization routines force the calling PE to halt execution until a symmetric data object is changed by a remote write or atomic swap issued by another PE.

8.14.1 shmem_wait

Summary

These routines force the calling PE to wait until **var** is no longer equal to **value**.

Synopsis

C/C++:

```
void shmem_short_wait(short *var, short value);
void shmem_int_wait(int *var, int value);
void shmem_long_wait(long *var, long value);
void shmem_longlong_wait(long long *var, long long value);
void shmem_wait(long *ivar, long value);
```

Fortran:

```
CALL SHMEM_INT4_WAIT(var, value)
CALL SHMEM_INT8_WAIT(var, value)
CALL SHMEM_WAIT(var, value)
```

Parameters

var The symmetric data object to be monitored on the calling PE.

value Value to be compared against the value at address **var**.

Constraints

- **var** must be the address of a symmetric data object.
- If using C/C++, the type of **var** must match that implied in the Synopsis section. When calling from Fortran, the data type of **var** must be as follows:
  - For **SHMEM_INT4_WAIT()**, **var** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_WAIT()**, **var** must be of type Integer, with element size of 8 bytes.
- **value** must be the same type as **var**.

Effect

A call to any **shmem_wait()** routine does not return until some other processor makes the value at address **var** not equal to **value**.

Return Values

None.

8.14.2 shmem_wait_until

Summary

These routines force the calling PE to wait until the condition indicated by **cond** and **value** is satisfied.

Synopsis

C/C++:

```
void shmem_short_wait_until(short *var, int cmp, short value);
void shmem_int_wait_until(int *var, int cmp, int value);
void shmem_long_wait_until(long *var, int cmp, long  value);
void shmem_longlong_wait_until(long long *var, int cmp, long long value);
void shmem_wait_until(long *ivar, int cmp, long value);
```

Fortran:

```
INTEGER cmp

CALL SHMEM_INT4_WAIT_UNTIL(var, cmp, value)
CALL SHMEM_INT8_WAIT_UNTIL(var, cmp, value)
CALL SHMEM_WAIT_UNTIL(var, cmp, value)
```

Parameters

var The symmetric data object to be monitored on the calling PE.

cmp Indicates how to compare value at address **var** and **value**. The following are the allowed compare operations:

SHMEM_CMP_EQ Equal
SHMEM_CMP_NE Not equal
SHMEM_CMP_GT Greater than
SHMEM_CMP_LE Less than or equal to
SHMEM_CMP_LT Less than

SHMEM_CMP_GE Greater than or equal to

value Value to be compared against the value at address **var**.

Constraints

- **var** must be the address of a symmetric data object.
- If using C/C++, the type of **var** must match that implied in the Synopsis section. When calling from Fortran, the data type of **var** must be as follows:
  - For **SHMEM_INT4_WAIT_UNTIL()**, **var** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_WAIT_UNTIL()**, **var** must be of type Integer, with element size of 8 bytes.
- **value** must be the same type as **var**.
- If using C/C++ it must be of integer type. If you are using Fortran, **cmp** must be of default kind.

Effect

A call to any **shmem_wait_until()** routine does not return until some other processor changes the value at address **var** to satisfy the condition implied by **cmp** and **value**.

Return Values

None.

8.15 Barrier Synchronization Routines

8.15.1 shmem_barrier_all

Summary

Suspends the execution of the calling PE until all other PEs issue a call to this particular **shmem_barrier_all()** statement.

Synopsis

C/C++:

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

Fortran:

```
CALL SHMEM_BARRIER_ALL()
```

Parameters

None.

Constraints

- All PEs must call this routine at the same point of the execution.

Effect

The **shmem_barrier_all()** routine does not return until all other PEs have entered this routine at the same point of the execution path.

Prior to synchronizing with other PEs, **shmem_barrier_all()** ensures completion of all previously issued local memory stores and remote memory updates issued via OpenSHMEM functions such as **shmem_put32**.

Return Values

None.

8.15.2 shmem_barrier

Summary

Performs a barrier operation on a subset of processing elements (PEs).

Synopsis

C/C++:

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

Fortran:

```
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BARRIER_SYNC_SIZE)

CALL SHMEM_BARRIER(PE_start, logPE_stride, PE_size, pSync)
```

Parameters

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pSync A symmetric work array.

Constraints

- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- In C/C++, pSync must be type integer and of 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** (**SHMEM_SYNC_VALUE** in Fortran) before any of the PEs in the active set enter shmem_barrier the first time.

Effect

The **shmem_barrier()** routine does not return until the subset of PEs specified by **PE_start**, **logPE_stride** and **PE_size**, have entered this routine at the same point of the execution path.

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.

Return Values

None.

8.15.3 shmem_fence

Summary

Provides a separate ordering on the sequence of puts issued by this PE to each destination PE.

Synopsis

C/C++:

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

Fortran:

```
CALL SHMEM_FENCE
```

Parameters

None.

Constraints

None.

Effect

The **shmem_fence()** routine provides an ordering on the put operations issued by the calling PE prior to the call to **shmem_fence()** relative to the put operations issued by the calling PE following the call to **shmem_fence()**. It guarantees that all such prior put operations issued to a particular destination PE are fully written to the symmetric memory of that destination PE, before any such following put operations to that same destination PE are written to the symmetric memory of that destination PE.

Note that the ordering is provided separately on the sequences of puts from the calling PE to each distinct destination PE. The **shmem_quiet()** routine should be used instead if ordering of puts is required when multiple destination PEs are involved.

Return Values

None.

8.15.4 shmem_quiet

Summary

Provides an ordering on the sequence of puts issued by this PE across all destination PEs.

Synopsis

C/C++:

```
void shmem_quiet(void);
```

Fortran:

```
CALL SHMEM_QUIET
```

Parameters

None.

Constraints

None.

Effect

The **shmem_quiet()** routine provides an ordering on the put operations issued by the calling PE prior to the call to **shmem_quiet()** relative to memory load, memory store, put, get or synchronization operations issued by the calling PE following the call to **shmem_quiet()**. It guarantees that all such prior put operations are fully written to the symmetric memory of the destination PE and visible to all other PEs, before any such following memory load, memory store, put, get or synchronization operations are visible to any PE.

Return Values

None.

8.16 Reduction Routines

The OpenSHMEM reduction routines perform an associative binary operation across symmetric arrays on multiple PEs. See Figure 3 for a generalized diagram of reduction operations.

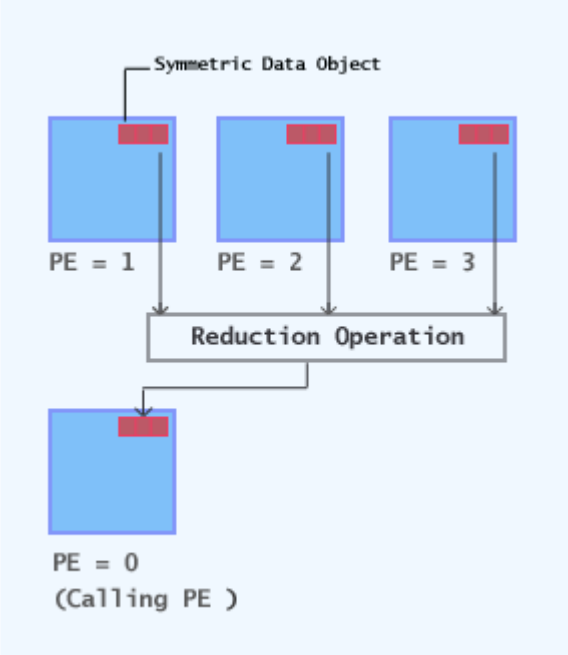


Fig. 3: General diagram of reduction operations

8.16.1 shmem_and

Summary

Performs a bitwise AND operation on symmetric arrays over the active set of PEs.

Synopsis

C/C++:

```
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_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);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- **target** and **source** must be the addresses of symmetric data objects.
- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:
  - For **SHMEM_INT4_AND_TO_ALL()**, **var** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_AND_TO_ALL()**, **var** must be of type Integer, with element size of 8 bytes.

- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

This routine returns the result of performing a bitwise AND operation on the **source** data object of every PE in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the bitwise AND operation and the elements of the **pSync** array will be restored to their initial values.

8.16.2 shmem_or

Summary

Performs a bitwise OR operation on symmetric arrays over the active set of PEs.

Synopsis

C/C++:

```
void shmem_short_or_to_all(short *target, short *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_or_to_all(int *target, int *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_or_to_all(long *target, long *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_or_to_all(long long *target, long long *source, int
   nreduce, int PE_start, int logPE_stride, int PE_size, long long *pWrk,
   long *pSync);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- **target** and **source** must be the addresses of symmetric data objects.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:
  - For **SHMEM_INT4_OR_TO_ALL()**, **target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_OR_TO_ALL()**, **target** must be of type Integer, with element size of 8 bytes.

- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

This routine returns the result of performing a bitwise OR operation on the **source** data object of every PE in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the bitwise OR operation and the elements of the **pSync** array will be restored to their initial values.

8.16.3 shmem_xor

Summary

Performs a bitwise XOR operation on symmetric arrays over the active set of PEs. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**.

Synopsis

C/C++:

```
void shmem_short_xor_to_all(short *target, short *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_xor_to_all(int *target, int *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_xor_to_all(long *target, long *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_xor_to_all(long long *target, long long *source, int
   nreduce, int PE_start, int logPE_stride, int PE_size, long long *pWrk,
   long *pSync);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
CALL SHMEM_COMP4_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_COMP8_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- **target** and **source** must be the addresses of symmetric data objects.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:

  - For **SHMEM_INT4_XOR_TO_ALL()**, **target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_XOR_TO_ALL()**, **target** must be of type Integer, with element size of 8 bytes.

- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

This routine returns the result of performing a bitwise XOR operation on the **source** data object of every PE in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the bitwise XOR operation and the elements of the **pSync** array will be restored to their initial values.

8.16.4 shmem_max

Summary

Computes the maximum value of the **source** symmetric array over the active set of PEs.

Synopsis

C/C++:

```
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_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_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_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_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_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);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- **target** and **source** must be the addresses of symmetric data objects.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:
  - For **SHMEM_INT4_MAX_TO_ALL()**, **target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_MAX_TO_ALL()**, **target** must be of type Integer, with element size of 8 bytes.
  - For **SHMEM_REAL4_MAX_TO_ALL()**, **target** must be of type Real, with element size of 4 bytes.
  - For **SHMEM_REAL8_MAX_TO_ALL()**, **target** must be of type Real, with element size of 8 bytes.
  - For **SHMEM_REAL16_MAX_TO_ALL()**, **target** must be of type Real, with element size of 16 bytes.s
- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

The max reduction routines determine the element with the highest value in array **source** across all PEs in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**. The results of the reduction must be stored at address **target** on all PEs of the active set.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the operation and the elements of the **pSync** array will be restored to their initial values.

8.16.5 shmem_min

Summary

Computes the minimum value of symmetric arrays over the active set of PEs. The active set of PEs is defined by the triplet **PE_start**, **logPE_stride** and **PE_size**.

Synopsis

C/C++:

```
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_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_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_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_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_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);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- **target** and **source** must be the addresses of symmetric data objects.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:
  - For **SHMEM_INT4_MIN_TO_ALL()**, **target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_MIN_TO_ALL()**, **target** must be of type Integer, with element size of 8 bytes.
  - For **SHMEM_REAL4_MIN_TO_ALL()**, **target** must be of type Real, with element size of 4 bytes.
  - For **SHMEM_REAL8_MIN_TO_ALL()**, **target** must be of type Real, with element size of 8 bytes.
  - For **SHMEM_REAL16_MIN_TO_ALL()**, **target** must be of type Real, with element size of 16 bytes.
- **source** must be the same type as **target**.

- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

The min reduction routines determine the element with the lowest value in array **source** across all PEs in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**. The results of the reduction must be stored at address **target** on all PEs of the active set.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the operation and the elements of the **pSync** array will be restored to their initial values.

8.16.6 shmem_sum

Summary

Computes the summation of symmetric arrays over the active set of PEs.

Synopsis

C/C++:

```
void shmem_short_sum_to_all(short *target, short *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_int_sum_to_all(int *target, int *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_sum_to_all(long *target, long *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_float_sum_to_all(float *target, float *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_double_sum_to_all(double *target, double *source, int nreduce,
   int PE_start, int logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_longlong_sum_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_longdouble_sum_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_complexf_sum_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_complexd_sum_to_all(double complex *target, double complex
   *source, int nreduce, int PE_start, int logPE_stride, int PE_size,
   double complex *pWrk, long *pSync);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- **target** and **source** must be the addresses of symmetric data objects.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:
  - For **SHMEM_INT4_SUM_TO_ALL()**, **target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_SUM_TO_ALL()**, **target** must be of type Integer, with element size of 8 bytes.
  - For **SHMEM_REAL4_SUM_TO_ALL()**, **target** must be of type Real, with element size of 4 bytes.
  - For **SHMEM_REAL8_SUM_TO_ALL()**, **target** must be of type Real, with element size of 8 bytes.
  - For **SHMEM_REAL16_SUM_TO_ALL()**, **target** must be of type Real, with element size of 16 bytes.
- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

The sum reduction routines compute the summation of **nreduce** elements in array **source** across all PEs in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**. The results of the reduction must be stored at address **target** on all PEs of the active set.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the operation and the elements of the **pSync** array will be restored to their initial values.

8.16.7 shmem_prod

Summary

Computes the product of symmetric arrays over the active set of PEs.

Synopsis

C/C++:

```
void shmem_short_prod_to_all(short *target, short *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_int_prod_to_all(int *target, int *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_prod_to_all(long *target, long *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_float_prod_to_all(float *target, float *source, int nreduce, int
   PE_start, int logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_double_prod_to_all(double *target, double *source, int nreduce,
   int PE_start, int logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_longlong_prod_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_longdouble_prod_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_complexf_prod_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_complexd_prod_to_all(double complex *target, double complex
   *source, int nreduce, int PE_start, int logPE_stride, int PE_size,
   double complex *pWrk, long *pSync);
```

Fortran:

```
INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

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)
```

Parameters

target Address of the symmetric data object where to store the results of the reduction operation.

source Address of the symmetric data object that contains the elements for each separate reduction operation.

nreduce The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pWrk A symmetric work array.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the reduction.
- **target** and **source** must be the addresses of symmetric data objects.
- If using C/C++, the type of **target** must match that implied in the Synopsis section. When calling from Fortran, the data type of **target** must be as follows:
  - For **SHMEM_INT4_PROD_TO_ALL()**, **target** must be of type Integer, with element size of 4 bytes.
  - For **SHMEM_INT8_PROD_TO_ALL()**, **target** must be of type Integer, with element size of 8 bytes.

  - For **SHMEM_REAL4_PROD_TO_ALL()**, **target** must be of type Real, with element size of 4 bytes.
  - For **SHMEM_REAL8_PROD_TO_ALL()**, **target** must be of type Real, with element size of 8 bytes.
  - For **SHMEM_REAL16_PROD_TO_ALL()**, **target** must be of type Real, with element size of 16 bytes.

- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nreduce** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- 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 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.
- The **pWrk** and **pSync** arrays on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

The prod reduction routines compute the summation of **nreduce** elements in array **source** across all PEs in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**. The results of the reduction must be stored at address **target** on all PEs of the active set.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon completion, the **target** array will be updated with the result of the operation and the elements of the **pSync** array will be restored to their initial values.

8.17 Collect Routines

8.17.1 shmem_collect

Summary

Concatenates blocks of data from multiple processing elements (PEs) to an array in every PE .

Synopsis

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:

```
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_FCOLLECT4(target, source, nelems, PE_start, logPE_stride,
   PE_size, pSync)
CALL SHMEM_FCOLLECT8(target, source, nelems, PE_start, logPE_stride,
   PE_size, pSync)
```

Parameters

target Address of the symmetric data object where to store the results of the collect operation.

source Address of the symmetric data object that contains the data to be concatenated.

nelems The number of elements in the **target** and **source** arrays.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the concatenation. It must also be large enough to accept the results of the concatenation.
- **target** and **source** must be the addresses of symmetric data objects.
- Data types for target are as follows:
  - For **shmem_collect32(), shmem_fcollect32(), SHMEM_COLLECT4** and **SHMEM_FCOLLECT4**, **target** must be of type Integer, with element size of 32 bytes.
  - For **shmem_collect64(), shmem_fcollect64(), SHMEM_COLLECT8** and **SHMEM_FCOLLECT8**, **target** must be of type Integer, with element size of 64 bytes.
- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nelems** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- The **pSync** array on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

The collect routines concatenate **nelems** elements from array **source** across all PEs in the active set, and store the result in array **target**. The fcollect 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 calls a OpenSHMEM collective routine, undefined behavior results.

Notes

All OpenSHMEM reduction routines reset the values in pSync before they return, so a particular pSync buffer need only be initialized the first OpenSHMEM reduction routine. Be careful of 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 a 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. However, in the special case of reduction routines being called with the same active set, you can allocate two pSync and pWrk arrays and alternate between them on successive calls.

Return Values

Upon successful completion of these routines, **target** will have the result of the concatenation.

8.18 Broadcast Routines

8.18.1 shmem_broadcast

Summary

Copy a data object from a designated PE to a target location on all other PEs of the active set. See Figure 4 for a diagram of a simple broadcast operation.

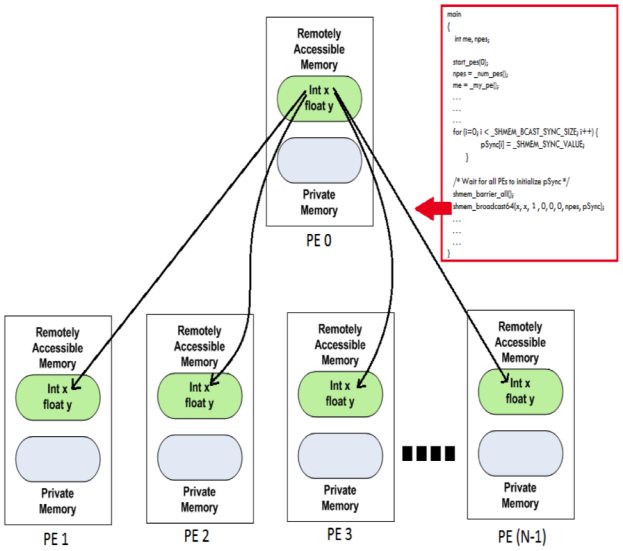


Fig. 4: Diagram of a simple broadcast operation.

Synopsis

C/C++:

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

Fortran:

```
INTEGER nelems
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)

CALL SHMEM_BROADCAST4(target, source, nelems, PE_root, PE_start,
   logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST8(target, source, nelems, PE_root, PE_start,
   logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST32(target, source, nelems, PE_root, PE_start,
   logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST64(target, source, nelems, PE_root, PE_start,
   logPE_stride, PE_size, pSync)
```

Parameters

target Address of the symmetric data object in which to store the data.

source Address of the symmetric data object that contains data to be copied.

nelems The number of elements in the **target** and **source** arrays.

PE_root The PE from which data will be copied.

PE_start The lowest PE number of the active set of PEs.

logPE_stride The log (base 2) of the stride between consecutive PE numbers in the active set.

PE_size The number of PEs in the active set.

pSync A symmetric work array.

Constraints

- The **target** array on all PEs in the active set must be ready to accept the results of the broadcast.
- **source** and **target** must be the addresses of symmetric data objects.
- Data types for target are as follows:
  - For **shmem_broadcast8()** and **shmem_broadcast64()**, **target** may be of any non character type that has an element size of 64 bits. No Fortran derived types or C/C++ structures are allowed.
  - For **shmem_broadcast32()**, **target** may be of any non character type that has an element size of 32 bits. No Fortran derived types or C/C++ structures are allowed.
  - For **shmem_broadcast4()**, **target** may be of any non character type that has an element size of 32 bits.
- **source** must be the same type as **target**.
- **source** and **target** may be the same array, but they must not be overlapping arrays.
- If using C/C++, **nelems** must be of type integer. If you are using Fortran, it must be a default integer value.
- **PE_root** is a zero-based integer count into the active set: $0 \leq PE\_root < PE\_size$. If you are using Fortran, it must be a default integer value.

- If using C/C++, **PE_start** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be greater than or equal to zero.
- If using C/C++, **logPE_stride** must be of type integer. If you are using Fortran, it must be a default integer value.
- If using C/C++, **PE_size** must be of type integer. If you are using Fortran, it must be a default integer value. Its value must be less than or equal to the total number of PEs minus one.
- In C/C++, **pSync** must be of type long and size **_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 the broadcast.
- The **pSync** array on all PEs in the active set must not be in use from a prior call to a collective OpenSHMEM routine.
- This routine must be called by all the PEs in the active set at the same point of the execution path; otherwise undefined behavior results.

Effect

The broadcast routines write the data at address **source** of the PE specified by **PE_root** to address **target** on all other PEs in the active set. The active set of PEs is defined by the triple **PE_start**, **logPE_stride** and **PE_size**. The data is not copied to the **target** address on the PE specified by **PE_root**.

Before returning, the broadcast routines ensure that the elements of the **pSync** array are restored to their initial values.

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 a OpenSHMEM collective routine, undefined behavior results.

Notes

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. However, in the special case of OpenSHMEM routines being called with the same active set, you can allocate two pSync arrays and alternate between them on successive calls.

Return Values

None.

8.19 Lock Routines

The OpenSHMEM lock routines manage mutual exclusion memory locks. These routines are appropriate for protecting a critical region from simultaneous update by multiple PEs.

8.19.1 shmem_set_lock

Summary

Sets a mutual exclusion memory lock after it is no longer in use by another process.

Synopsis

C/C++:

```
void shmem_set_lock(long *lock);
```

Fortran:

```
INTEGER lock

CALL SHMEM_SET_LOCK(lock)
```

Parameters

lock Address of a symmetric data object that is a scalar variable or an array of length 1.

Constraints

- The value at address **lock** must be set to 0 on all PEs prior to the first use.
- If using C/C++, **lock** must be of type integer. If you are using Fortran, it must be of default kind.

Effect

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.

Return Values

None.

8.19.2 shmem_clear_lock

Summary

Releases a lock previously set by the calling PE.

Synopsis

C/C++:

```
void shmem_clear_lock(long *lock);
```

Fortran:

```
INTEGER lock

CALL SHMEM_CLEAR_LOCK(lock)
```

Parameters

lock Address of a symmetric data object that is a scalar variable or an array of length 1.

Constraints

- If using C/C++, **lock** must be of type integer. If you are using Fortran, it must be of default kind.
- The lock can only be released by the PE that previously set the lock.

Effect

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.

Return Values

None.

8.19.3 shmem_test_lock

Summary

Sets a mutual exclusion lock only if it is currently cleared.

Synopsis

C/C++:

```
int shmem_test_lock(long *lock);
```

Fortran:

```
INTEGER lock, SHMEM_TEST_LOCK

I = SHMEM_SET_LOCK(lock)
```

Parameters

lock Address of a symmetric data object that is a scalar variable or an array of length 1.

Constraints

- If using C/C++, **lock** must be of type integer. If you are using Fortran, it must be of default kind.

Effect

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.

Return Values

0 The lock was originally cleared and this call was able to set the lock.

1 The lock had been set and the call returned without waiting to set the lock.

Note to implementers: the lock variable is initialized to 0 everywhere. Once set, the value of the lock should be treated as opaque to allow implementations freedom to optimize lock structures, e.g. for specific hardware operations.

8.20 Cache Management Routines

The OpenSHMEM specification defines these routines to maintain application compatibility.

The cache management routines allow the OpenSHMEM implementation to take advantage of hardware cache to improve application performance.

8.20.1 shmem_set_cache_inv

Summary

Enables automatic cache coherency mode.

Synopsis

C/C++:

```
void shmem_set_cache_inv(void);
```

Fortran:

```
CALL SHMEM_SET_CACHE_INV()
```

Parameters

None.

Constraints

None.

Effect

Enables the OpenSHMEM API to automatically decide the best strategy for cache coherency.

Return Values

None.

8.20.2 shmem_set_cache_line_inv

Summary

Enable cache coherency for a specific object only.

Synopsis

C/C++:

```
void shmem_set_cache_line_inv(void *target);
```

Fortran:

```
CALL SHMEM_SET_CACHE_LINE_INV(target)
```

Parameters

target Address of the symmetric data object.

Constraints

- If using C/C++, **target** can be of any non character type. If you are using Fortran, it can be of any kind.

Effect

Enables automatic cache coherency mode for the cache line associated with the address of **target** only.

Return Values

None.

8.20.3 shmem_clear_cache_inv

Summary

Disable cache coherency.

Synopsis

C/C++:

```
void shmem_clear_cache_inv(void);
```

Fortran:

```
CALL SHMEM_CLEAR_CACHE_INV()
```

Parameters

None.

Constraints

None.

Effect

Disables automatic cache coherency mode previously enabled by **shmem_set_cache_inv()** or **shmem_set_cache_line_inv()**.

Return Values

None.

8.20.4 shmem_clear_cache_line_inv

Summary

Disable cache coherency.

Synopsis

C/C++:

```
void shmem_clear_cache_line_inv(void* target);
```

Fortran:

```
CALL SHMEM_CLEAR_CACHE_LINE_INV(target)
```

Parameters

None.

Constraints

None.

Effect

Disables automatic cache coherency mode for the cache line associated with the address of **target** only.

Return Values

None.

8.20.5 shmem_udcflush

Summary

Makes the entire user data cache coherent.

Synopsis

C/C++:

```
void shmem_udcflush(void);
```

Fortran:

```
CALL SHMEM_UDCFLUSH()
```

Parameters

None.

Constraints

None.

Effect

Makes the entire user data cache coherent.

Return Values

None.

8.20.6 shmem_udcflush_line

Summary

Enable cache coherency for a specified data object only.

Synopsis

C/C++:

```
void shmem_udcflush_line(void *target);
```

Fortran:

```
CALL SHMEM_UDCFLUSH_LINE(target)
```

Parameters

target Address of the symmetric data object.

Constraints

- If using C/C++, **target** can be of any non character type. If you are using Fortran, it can be of any kind.

Effect

Makes coherent the cache line that corresponds with the address specified by **target**.

Return Values

None.

9 Library Constants

9.1 Constants Related To Reduction Operations

9.1.1 SHMEM_BCAST_SYNC_SIZE, _SHMEM_BCAST_SYNC_SIZE

Length of the pSync arrays needed for broadcast operations. The value of this constant is implementation specific. Refer to the **Broadcast Routines** section under **Library Routines** for more information about the usage of this constant.

9.1.2 SHMEM_SYNC_VALUE, _SHMEM_SYNC_VALUE

Holds the value used to initialize the elements of pSync arrays. The value of this constant is implementation specific.

9.1.3 SHMEM_REDUCE_SYNC_SIZE, _SHMEM_REDUCE_SYNC_SIZE

Length of the work arrays needed for reduction operations. The value of this constant is implementation specific. Refer to the **Reduction Routines** section under **Library Routines** for more information about the usage of this constant.

9.1.4 SHMEM_BARRIER_SYNC_SIZE, _SHMEM_BARRIER_SYNC_SIZE

Length of the work array needed for barrier operations. The value of this constant is implementation specific. Refer to the **Barrier Synchronization Routines** section under **Library Routines** for more information about the usage of this constant.

9.1.5 SHMEM_COLLECT_SYNC_SIZE, _SHMEM_COLLECT_SYNC_SIZE

Length of the work array needed for collect operations. The value of this constant is implementation specific. Refer to the **Collect Routines** section under **Library Routines** for more information about the usage of this constant.

9.1.6 SHMEM_REDUCE_MIN_WRKDATA_SIZE, _SHMEM_REDUCE_MIN_WRKDATA_SIZE

Minimum length of work arrays used in various collective operations.

10 Writing OpenSHMEM Programs

10.1 Incorporating OpenSHMEM into Programs

C and C++ programs that use the OpenSHMEM library *must*

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

All Fortran OpenSHMEM programs *should*

```
include 'shmem.fh'
```

and Fortran OpenSHMEM programs that use constants defined by OpenSHMEM *must*

```
include 'shmem.fh'
```

10.1.1 Compatibility Note

Implementations *must* also provide these header files so that they can be referenced in C and C++ as

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

and in Fortran as

```
include 'mpp/shmem.fh'
```

for backward compatbility with OpensHMEM 1.0 and SGI SHMEM.

10.2 Initialization

An OpenSHMEM program must call **start_pes()** (See Section Subsection 8.1.1) before any other OpenSHMEM routine. If **start_pes()** is not called first, the subsequent behavior of OpenSHMEM is undefined. Calling **start_pes()** more than once has no subsequent effect. The parameter to **start_pes()** is ignored; the number of PEs is taken from the invoking environment.

A Environment Variables

Consistent with the SGI implementation of SHMEM, the OpenSHMEM specification currently provides a set of environment variables that allows users to affect run-time behavior:

Variable	Value	Function
SMA_VERSION	any	print the library version at start-up
SMA_INFO	any	print helpful text about all these environment variable
SMA_SYMMETRIC_SIZE	non-negative integer	number of bytes to allocate for symmetric heap
SMA_DEBUG	any	enable debugging messages

Implementations are free to define their own environment variables.

B Compiling and Running Applications

The OpenSHMEM specification is silent regarding how OpenSHMEM programs are compiled, linked and run. This section shows some examples of how wrapper programs could be utilized to compile and launch applications. The commands are styled after wrapper programs found in many MPI implementations.

B.1 Compilation

B.1.1 Applications written in C

Assuming that the 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
```

The program arguments for **oshcc** are:

<compiler options> Options understood by the underlying C compiler called by **oshcc**.

B.2 Applications written in C++

Assuming that the 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
```

The program arguments for **oshCC** are:

<compiler options> Options understood by the underlying C++ compiler called by **oshCC**.

B.3 Applications written in Fortran

Assuming that the 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
```

The program arguments for **oshfort** are:

<compiler options> Options understood by the underlying Fortran compiler called by **oshfort**.

C Running Applications

Assuming that the implementation provides a wrapper program named **oshrun,** to launch OpenSHMEM applications, the wrapper could be called as follows:

```
oshrun <additional options> -np <#> <program>
```

The program arguments for **oshrun** are:

<additional options> options passed to the underlying launcher

-np <#> The number of processing elements (PEs) to be used in the execution.

<program> The program executable to be launched.

D Examples

D.1 C Language Examples

Listing 1: Program that is a trivial Hello World.

```c
#include <shmem.h>

int main(int argc, char* argv[])
{
        int i, me, my_num_pes;
        /*
        ** Starts/Initializes SHMEM/OpenSHMEM
        */
        start_pes(0);
        /*
        ** Fetch the number or processes
        ** Some implementations use num_pes();
        */
        my_num_pes = _num_pes();
        /*
        ** Assign my process ID to me
        */
        me = _my_pe();
        printf("Hello World from %d of %d\n",me,my_num_pes);
        return 0;
}
```

Listing 2: Program that implements a Circular Shift.

```c
/* circular shift bbb into aaa */
#include <shmem.h>

int aaa, bbb;

int main (int argc, char * argv[])
{
        start_pes(0);
        shmem_int_get(&aaa, &bbb, 1,(_my_pe() + 1)% _num_pes());
        shmem_barrier_all();
}
```

Listing 3: Program that demonstrates the use of shmalloc.

```
/*
* OpenSHMEM program to allocate (shmalloc) symmetric memory (1 long integer),
* and then free it.  Success of allocation is untested.
*
* This program produces no output.
*/

#include <shmem.h>

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

Listing 4: Program that implements Ping.

```
/*
 * test if PE is accessible
 *
 */

#include <stdio.h>

#include <shmem.h>

int
main(void)
{
  int me, npes;

  setbuf(stdout, NULL);

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

  if (me == 0) {
    int i;
    for (i = 1; i < npes; i += 1) {
      printf("From %d: PE %d is ", me, i);
      printf("%s", shmem_pe_accessible(i) ? "" : "NOT ");
      printf("accessible\n");
    }
  }

  return 0;
}
```

Listing 5: Program that uses the MAX reduction.

```
/*
 * reduce [0,1,2] + _my_pe() across 4 PEs with MAX()
 *
 *
 */

#include <stdio.h>
#include <string.h>

#include <shmem.h>

long pSync[_SHMEM_BCAST_SYNC_SIZE];

#define N 3

long src[N];
long dst[N];
long pWrk[_SHMEM_REDUCE_SYNC_SIZE];

int
main(void)
{
  int i;

  for (i = 0; i < SHMEM_BCAST_SYNC_SIZE; i += 1) {
    pSync[i] = _SHMEM_SYNC_VALUE;
  }

  start_pes(0);

  for (i = 0; i < N; i += 1) {
    src[i] = _my_pe() + i;
  }
  shmem_barrier_all();

  shmem_long_max_to_all(dst, src, N, 0, 0, _num_pes(), pWrk, pSync);

  printf("%d/%d   dst =", _my_pe(), _num_pes() );
  for (i = 0; i < N; i+= 1) {
    printf(" %d", dst[i]);
  }
  printf("\n");

  return 0;
}
```

Listing 6: Program that makes use of strided puts.

```
/*
 * This program is an adaptation of examples found in the man pages
 * of SGI's SHMEM implementation.
 *
 * In this program, iput is used to select 5 elements from array source separated by
 * a stride of 2 and write them to array target using a stride of 1.
 *
 * Given the array source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
 * iput will select 5 elements from array source on PE 0, using a stride of 2:
 *
 * selected elements = { 1, 3, 5, 7, 9 }
 *
 * These elements will then be written to the array source on PE 1 using a stride of 1:
 *
 * target = { 1, 3, 5, 7, 9 }
 *
 */

#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];
  int me;

  start_pes(0);
  me = _my_pe();

  if (me == 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 (me == 1) {
    printf("target on PE %d is %hd %hd %hd %hd %hd\n", me,
           target[0], target[1], target[2],
           target[3], target[4] );
  }
  shmem_barrier_all();   /* sync before exiting */

  return 0;
}
```

Listing 7: Program that implements an ALL-2-ALL (header)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>


typedef   signed  char  int8 ;
typedef unsigned  char uint8 ;
typedef          short  int16;
typedef unsigned short uint16;
typedef            int  int32;
typedef unsigned   int uint32;
typedef           long  int64;
typedef unsigned  long uint64;

/* timing */

#include <sys/time.h>
#include <time.h>

/* wall-clock time */

static double wall(void)

{
  struct timeval tp;

  gettimeofday (&tp, NULL);
  return
    tp.tv_sec + tp.tv_usec/(double)1.0e6;
}

#include <sys/resource.h>

/* cpu + system time */

static double cpus(void)
{
  struct rusage ru;

  getrusage(RUSAGE_SELF,&ru);
  return
    (ru.ru_utime.tv_sec  + ru.ru_stime.tv_sec) +
    (ru.ru_utime.tv_usec + ru.ru_stime.tv_usec)/(double)1.0e6;
}

typedef struct {
  double accum_wall, accum_cpus;
  double start_wall, start_cpus;
  time_t init_time;
  char running;
} timer;

static void timer_clear (timer *t)
{
  t->accum_wall = t->accum_cpus = 0;
  t->start_wall = t->start_cpus = 0;
  t->running = 0;
}
```

```

static void timer_start (timer *t)
{
  t->start_wall = wall();
  t->start_cpus = cpus();
  t->running = 1;
}

static void timer_stop  (timer *t)
{
  if (t->running == 0)
    return;
  t->accum_cpus += cpus() - t->start_cpus;
  t->accum_wall += wall() - t->start_wall;
  t->running = 0;
}

static void timer_report (timer *t, double *pwall, double *pcpus,
                          int64 print)
{
  double w, c;

  w = t->accum_wall;
  c = t->accum_cpus;

  if (t->running) {
    c += cpus() - t->start_cpus;
    w += wall() - t->start_wall;
  }
  if (print) {
    printf ("%7.3f secs of wall clock time\n",     w);
    printf ("%7.3f secs of cpu and system time\n", c);
  }

  if (pwall)  *pwall = w;
  if (pcpus)  *pcpus = c;
}


/* some masking macros */

#define _ZERO64       0uL
#define _maskl(x)     (((x) == 0) ? _ZERO64   : ((~_ZERO64) << (64-(x))))
#define _maskr(x)     (((x) == 0) ? _ZERO64   : ((~_ZERO64) >> (64-(x))))
#define _mask(x)      (((x) < 64) ? _maskl(x) : _maskr(2*64 - (x)))

/* PRNG */

#define _BR_RUNUP_      128L
#define _BR_LG_TABSZ_    7L
#define _BR_TABSZ_      (1L<<_BR_LG_TABSZ_)

typedef struct {
  uint64 hi, lo, ind;
  uint64 tab[_BR_TABSZ_];
} brand_t;

#define _BR_64STEP_(H,L,A,B) {\
  uint64 x;\
```

```
  x = H ^ (H << A) ^ (L >> (64-A));\
  H = L | (x >> (B-64));\
  L = x << (128 - B);\
}

static uint64 brand (brand_t *p) {
  uint64 hi=p->hi, lo=p->lo, i=p->ind, ret;

  ret = p->tab[i];

  _BR_64STEP_(hi,lo,45,118);

  p->tab[i] = ret + hi;

  p->hi  = hi;
  p->lo  = lo;
  p->ind = hi & _maskr(_BR_LG_TABSZ_);

  return ret;
}

static void brand_init (brand_t *p, uint64 val)

{
  int64 i;
  uint64 hi, lo;

  hi = 0x9ccae22ed2c6e578uL ^ val;
  lo = 0xce4db5d70739bd22uL & _maskl(118-64);

  for (i = 0; i < 64; i++)
    _BR_64STEP_(hi,lo,33,118);

  for (i = 0; i < _BR_TABSZ_; i++) {
    _BR_64STEP_(hi,lo,33,118);
    p->tab[i] = hi;
  }
  p->ind = _BR_TABSZ_/2;
  p->hi  = hi;
  p->lo  = lo;

  for (i = 0; i < _BR_RUNUP_; i++)
    brand(p);
}

/* init / end subroutines */

/* prints information, initializes PRNG, returns number of iterations */

#define INIT_ST  "INIT>"
#define END_ST   "END>"
#define MAX_HOST 80L

static int64 bench_init (int argc, char *argv[], brand_t *br,
                         timer *t, char *more_args)
{
  uint64 seed;
  int64 niters;
  int i;
```

```
177   time_t c;
178   static char host[MAX_HOST];
179
180   if ((i = sizeof(void *)) != 8) {
181     printf ("error: sizeof(void *) = %d\n", i);
182     exit(1);
183   }
184   if ((i = sizeof(long)) != 8) {
185     printf ("error: sizeof(long) = %d\n", i);
186     exit(1);
187   }
188   if ((i = sizeof(int)) != 4) {
189     printf ("error: sizeof(int) = %d\n", i);
190     exit(1);
191   }
192
193   if (argc < 3) {
194     /* prog seed iters [... other args] */
195     printf ("Usage:\t%s seed iters %s\n",
196             argv[0], (more_args != NULL) ? more_args : "");
197     exit(0);
198   }
199
200   printf ("\n==================================================\n\n");
201
202   /* print start time of day */
203   time (&c);
204   printf ("%s %s started at:  %s", INIT_ST, argv[0], ctime(&c));
205   t->init_time = c;
206
207   gethostname (host, MAX_HOST);
208   printf ("%s host machine is %s\n", INIT_ST, host);
209
210   printf ("%s program built on %s @ %s\n",
211           INIT_ST,  __DATE__, __TIME__);
212
213   seed   = atol (argv[1]);
214   niters = atol (argv[2]);
215
216   printf ("%s seed is %ld   niters is %ld\n", INIT_ST, seed, niters);
217   if (argc > 3) {
218     printf ("%s other args: ", INIT_ST);
219     argv += 3;
220     while (*argv)
221       printf (" %s", *argv++);
222     printf ("\n");
223   }
224
225   if (br != NULL)
226     brand_init (br, seed);
227
228   if (t != NULL)
229     timer_clear (t);
230
231   printf ("\n");
232
233   return niters;
234 }
235
```

```
236 static void bench_end (timer *t, int64 iters, char *work)
237 {
238   time_t c;
239   double wall, cpus, rate;
240 
241   printf ("\n");
242 
243   /* print end time of day */
244   time (&c);
245   printf ("%s ended at:  %s", END_ST, ctime(&c));
246   c = c - t->init_time;
247   printf ("%s elapsed time is %d seconds\n", END_ST, c);
248 
249   if (t != NULL) {
250     timer_report(t, &wall, &cpus, 0);
251 
252     printf ("%s %7.3f secs of wall time      ",
253             END_ST, wall);
254     if (c <= 0) c = 1;
255     printf ("%7.3f%% of value reported by time()\n", wall/c*100.);
256 
257     if (wall <= 0)  wall = 0.0001;
258     printf ("%s %7.3f secs of cpu+sys time   utilization = %5.3f%%\n",
259             END_ST, cpus, cpus/wall*100.);
260 
261     if (cpus > (wall+.01))
262       printf ("this result is suspicious since cpu+system > wall\n");
263     if ((iters > 0) && (work != NULL)) {
264       const char *units[4] = {"", "K", "M", "G"};
265       int i = 0;
266 
267       rate = iters/wall;
268       while (i < 3) {
269         if (rate > 999.999) {
270           rate /= 1024.;
271           i++;
272         }
273         else
274           break;
275       }
276 
277       printf ("%s %8.4f %s %s per second\n",
278               END_ST, rate, units[i], work);
279     }
280   }
281 }
```

Listing 8: Program that implements an ALL-2-ALL (main)

```
/* CVS info   */
/* $RCSfile: all2all_main.c,v $  */
/*
* Purpose: all2all.c copies data from one half of a table to the other
*          half of the table.
*
*
*   Date      Description
*
*             all2all has been modified to automatically compare cksum results
*             for 128 processors at run time and to print an error message
*             if there is a discrepancy.
*             In the future additional error checking for any number of processors
*             will be done.
*             added memset(tab,0,tsize)
*
* Preprocessor DEFINED Variables:
*       1. This benchmark will automatically verify checksums unless CHECKOFF is
*       defined in the makefile flags. To turn off the automated check specify
*       -DCHECKOFF in the makefile flags and recompile.
*       (MUST define CHECKOFF if not using 128 processors.)
*       2. If additional timing info is needed for debugging specify -DPTIMES in
*       the makfile flags.
*/

#include <string.h>
#include "all2all.h"

int64 SELF, SIZE;

#if 0
int64 known_v[] = {
 0x889d1f6f6b165117,
 0xc2597eee7a77503b,
 0x9fde67a85fec3140,
 0x98218560b0e2fcad,
 0x77970e91ec2ae92f,
 0xd7c257a76e652480,
 0xfae8fc3473e44bd7,
 0xae70524b190b97d1,
 0xbd3481e6d55c2587,
 0x92b1e34c9a63c162,
 0xd53483207d373375,
 0x818b5ae39e15de0c,
 0xa10c2c69b3441650,
 0x3213b203ef570cfe,
 0x953cacafbc6694af,
 0x0435c6359cfeac6a,
 0x0107162b374ac090,
 0x3b4579d543eb131e,
 0x1f46dbcd8e23ca22,
 0x4f99bd5b1c45bff2,
 0x69872eca2dd09002,
 0x5a10168c91da8c2e,
 0xfb7842751192f1bf,
 0x42d182c4447097fe,
 0xacdb47e7a6c94a44,
 0x91fb985dbdd6e93b,
```

```
 0x4796404dd92f2c3a,
 0xcda282a270d3610f,
 0x29d786ca8abdaf09,
 0x3f9af62d5a02bdc6,
 0x513eb2b11ab80a05,
 0x59a32e0cc53f2c3d,
 0x5b22688cc292ee8c,
 0xd7076df7f4c3b35b,
 0x3dcf8e920a889b72,
 0x6cf0fe53b376b881
};

#endif
int gv = 0;

/* Set up for one iteration only.*/
int64 ckv[3] =  {
   0x156a0e1af0914226,
   0xa70ebc57a39fd98d,
   0x1513f274d76734c6,
};

uint64 do_cksum        (uint64 *arr, int64 len)
{
  int64 i, cksum;

  // compute src cksum
  for (i = cksum = 0; i < len; i++)
    cksum += arr[i];
  return accum_long (cksum);
}

int main (int argc, char *argv[])
{
  static char cvs_info[] = "BMkGRP $Date: $ $Revision: $ $RCSfile: all2all_main.c,v $ $Name: $";

  int itr;
  int idx;
  brand_t br;
  timer t, t0, t1;
  double nsec;

  double total_time = 0.0;

  int status = 0;

  int64 i, seed, arg, msize, tsize, len, oldsize=0, rep, cksum;
  uint64 *tab=NULL;

  start_pes(0);
  SELF=_my_pe();
  SIZE=_n_pes();

  if (argc < 5) {
    if (SELF == 0)
      fprintf (stderr, "Usage:\t%s seed msg_size(B) table_size(MB) rep_cnt "
               "[ms2 ts2 rc2 ..]\n", argv[0]);
      status = 1;
    goto DONE;
```

```
118   }
119   seed = atol (argv[1]);
120   if (SELF == 0)
121     printf ("base seed is %ld\n", seed);
122   seed += SELF << 32;
123   brand_init (&br, seed);  // seed uniquely per PE
124
125   arg = 2;
126
127   while (arg < argc) {
128
129
130     msize = atol (argv[arg++]);               if (arg >= argc)  break;
131     /* Table size * 1 million. */
132     tsize = atol (argv[arg++]) * (1L << 20);  if (arg >= argc)  break;
133     //rep   = atol (argv[arg++]);
134     rep   = 1;
135     arg++;
136
137     if (SELF == 0)  printf ("tsize = %ldMB  msize = %dB\n",
138                             tsize/(1L<<20), msize);
139     if (msize < sizeof(long)) {
140       if (SELF == 0)  printf ("msize too short!\n");
141       //status = 1;
142       goto DONE;
143     }
144     //itr=0;
145
146     idx = 0;
147
148     switch(SIZE){
149       case 2:
150         idx = 0;
151         break;
152       case 4:
153         idx = 1;
154         break;
155       case 8:
156         idx = 2;
157         break;
158       default:
159         fprintf(stderr,"warning, check sum for (%d) pes not supported.\n",
160                         SIZE);
161     }
162
163     while (rep-- > 0) {
164
165       /* START TIMING */
166       //timer_clear (&t0);
167       //timer_clear (&t1);
168       //timer_start (&t0);
169
170       if ((tab == NULL) || (tsize > oldsize)) {
171         if (tab != NULL) {
172           dram_shfree (tab);
173           oldsize = 0;
174         }
175         if (SELF == 0)  printf ("trying dram_shmalloc of %ld bytes\n", tsize);
176         tab = (uint64 *) dram_shmalloc (tsize);
```

```
177
178         if (tab == NULL) {
179           if (SELF == 0)  printf ("dram_shmalloc failed!\n");
180           status = 1;
181           goto DONE;
182
183         }
184         oldsize = tsize;
185       }
186
187       // length in words
188       len = tsize / sizeof(uint64);
189
190       // important to init table
191       // to ensure cksum consistency on different platforms
192       memset(tab,0,tsize);
193
194       for (i = 0; i < len; i+=64){
195         tab[i] = brand(&br);
196       }
197
198       // we'll have destination/source arrays each of half size
199       len /= 2;
200
201       //timer_stop (&t0);
202       // source checksum
203       cksum = do_cksum (&tab[len], len);
204       if (SELF == 0)  printf ("cksum is %016lx\n", cksum);
205       if (SELF == 0){
206         //if(cksum!=ckv[itr++]){
207         /* Set up for one iteration only. */
208         if(cksum!=ckv[idx]){
209          printf ("cksum  %016lx != ckv[%d] %016x\n",cksum,idx,ckv[idx]);
210           gexit(1);
211         }
212
213       }
214
215       //timer_start (&t1);
216       len = do_all2all (&tab[0], &tab[len], len, msize/sizeof(uint64));
217
218       shmem_barrier_all();
219
220       //timer_stop (&t1);
221       /* END TIMING */
222 #if 0
223
224       // dest checksum
225       i = do_cksum (&tab[0], len);
226       if (i != cksum) {
227         printf ("PE %4ld  ERROR: %016lx != %016lx\n", SIZE, i, cksum);
228         status = 1;
229         goto DONE;
230       }
231
232 #ifndef CHECKOFF
233       if (i != known_v[gv]) {
234         printf ("CHECKSUM PE %4ld  ERROR: %016lx != %016lx\n", SIZE, i, known_v[gv]);
235         status = 1;
```

```
236         goto DONE;
237       }
238       gv++;
239 #endif
240 
241 
242       //t.accum_wall = t0.accum_wall + t1.accum_wall;
243       //t.accum_cpus = t0.accum_cpus + t1.accum_cpus;
244 
245 
246       /*if (SELF == 0) {
247 
248 #ifdef PTIMES
249         printf ("%8.3f %8.3f\n",    t0.accum_wall , t1.accum_wall);
250         printf ("%8.3f %8.3f\n",    t0.accum_cpus , t1.accum_cpus);
251 #endif
252         printf ("wall reports %8.3f secs  cpus report %8.3f secs\n",
253                 t.accum_wall, t.accum_cpus);
254         nsec = MAX(t.accum_wall, t.accum_cpus);
255         total_time += nsec;
256         if (nsec > 0)
257           printf ("%8.3f MB/sec with %ld bytes transfers\n",
258                   len*sizeof(uint64)/(double)(1L<<20)/nsec, msize);
259       }*/
260 #endif
261     }
262     //if (SELF == 0)
263       //printf ("\n");
264   }
265   //if (SELF == 0)
266   //{
267      //printf ("total time = %14.9f\n", total_time);
268 
269   //}
270 
271  DONE:
272   shmem_barrier_all();
273   return status;
274 }
```

Listing 9: Program that implements an ALL-2-ALL (subs)

```
#include "bench.h"
#include <shmem.h>

#define PERM(ME,TOT,ITER)  ((ME)^(ITER))         // ok if 2^n pes

#define MAX(A,B)  (((A)>(B)) ? (A) : (B))
#define MIN(A,B)  (((A)<(B)) ? (A) : (B))

int64  do_all2all      (uint64 *dst, uint64 *src, int64 len, int64 nwrd);
int64  accum_long      (int64 val);

extern int64 SELF, SIZE;

/* returns words sent per PE */

int64  do_all2all      (uint64 *dst, uint64 *src, int64 len, int64 nwrd)
{
static char cvs_info[] = "BMKGRP $Date: $ $Revision: $ $RCSfile: all2all.c,v $ $Name: $";

  int64 i, j, pe;

  len = len - (len % (nwrd * SIZE));  // force even multiple
  for (i = 0; i < len; i+=SIZE*nwrd) {
    shmem_barrier_all();
    for (j = 0; j < SIZE; j++) {
      pe = PERM(SELF,SIZE,j);
      /* shmem_put (&dst[i + SELF*nwrd], &src[i + pe*nwrd], nwrd, pe);*/
      shmem_put64 (&dst[i + SELF*nwrd], &src[i + pe*nwrd], nwrd, pe);
    }
  }
  return len;
}

int64  accum_long      (int64 val)

{
  int64 i;
  static int64 target, source, init=0;
  static int64 Sync[_SHMEM_REDUCE_SYNC_SIZE];
  static int64 Work[2 + _SHMEM_REDUCE_MIN_WRKDATA_SIZE];

  if (! init) {
    /* need to initialize Sync first time around */
    for(i = 0; i < _SHMEM_REDUCE_SYNC_SIZE; i++)
      Sync[i] = _SHMEM_SYNC_VALUE;
    init = 1;
  }
  source = val;
  shmem_barrier_all();

  shmem_long_sum_to_all (&target, &source, 1, 0, 0, SIZE, Work, Sync);

  shmem_barrier_all();
  return target;
}
```

Listing 10: Program that computes Pi

```
/*
 * This file is distributed as part of GatorSHMEM, a project of the HCS
 * Research Lab / CHREC at the University of Florida.
 *
 * Copyright (c) 2005-2010, the University of Florida.
 * All rights reserved.
 *
 * Modified by SPoole from ORNL to be OpenSHMEM V1.0 compliant
 * and work with other architectures.
 *
 */
#include <math.h>

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

#define M_PI_2     1.57079632679489661923
#define TRIES      1000000000


double timerval()
{
        struct timeval st;
        gettimeofday ( &st, NULL);
        return st.tv_sec + st.tv_usec * 1e-6;
}


int main(int argc, char *argv[])
{
// 1. get random [0, 1] ==>
// 2. get random [0, pi/2] ==>theta
// 3. hit X < sin(theta)
// 4. 2/pi = hit/tries.

  double X, Theta, My_pi;
  double Tstart, Tend;
  int i, total, hit=0;
  int *buf, my_mem;
  int rank, numprocs, num_of_procs;

  num_of_procs = atoi(argv[1]);

  start_pes (0);

  numprocs = _num_pes();
  rank     = _my_pe();

  my_mem = (sizeof(int) * numprocs);
  buf = shmalloc(my_mem);

  srand( (unsigned int) time(NULL));

  if ( rank == 0 ) {
    printf("pi is %f\n", M_PI_2 );
    printf("sin(pi/2) is %f\n", sin(M_PI_2));
```

```
59      fflush(stdout);
60    }
61
62    Tstart = timerval();
63
64    if ( rank != 0 ) {
65     total = TRIES/(numprocs-1);
66     if (rank == 1)
67       total += TRIES % (numprocs-1);
68
69     srand( (unsigned int) time(NULL));
70
71     for ( i = 0; i < total ; i++){
72     X = rand();
73     X = X/RAND_MAX;
74
75     Theta = rand();
76     Theta = ( M_PI_2 ) * (Theta/RAND_MAX);
77     if ( X < sin(Theta))
78       hit++;
79     }
80     buf[0] = hit;
81    }
82
83    shmem_barrier_all();
84
85    if ( rank == 0 )
86      for ( i = 1; i < numprocs; i++) {
87        shmem_getmem(buf, buf, sizeof(int), i);
88        hit += buf[0];
89        printf("from node(%d), getmem buf\t %d, so hit is\t %d\n", i, buf[0], hit);
90      }
91
92    shmem_barrier_all();
93
94    if ( rank == 0 ) {
95    My_pi = 2 * ( 1 / ( ((double)hit)/TRIES ) );
96    Tend = timerval();
97    printf("Hit is : %d :: Total is %d \n",hit, TRIES);
98    printf("My pi is %.16f \n", My_pi);
99    printf("Elapsed time is %f \n", Tend - Tstart);
100   }
101
102   shmem_barrier_all();
103
104   return 0;
105 }
```

D.2 Fortran Language Examples

Listing 11: Hello World program

```
program whoami

  include 'shmem.fh'

  integer npes, me
  character*32 h

  call start_pes(0)

  npes = num_pes()
  me = my_pe()
  call hostnm(h)

  print *, h, 'I am ', me, ' of ', npes

end program whoami
```

E Glossary

E.1 OpenSHMEM Concepts

processing element	One of the processors involved in the execution of an OpenSHMEM application. Called “PE” for short.
PE number	Integer value used to identify a processing element.
active set of PEs	The group of PEs involved in the execution of a collective operation.
OpenSHMEM program	Program that makes use of routines in the OpenSHMEM library.
put operation	An operation that places data on a remote processing element.
get operation	An operation that retrieves data from a remote processing element.
remote pointer	A pointer that allows to directly reference a data object that is stored in a remote processing element.
reduction	Performs an associative binary operation across a set of values on multiple PEs.
broadcast	A collective operation that places a data object available on a “root” PE, onto all other PEs in the broadcast’s active set.
barrier	A collective synchronization mechanism. PEs in the active set cannot leave the barrier routine until all of those PEs have reached the barrier.

E.2 Data Terminology

cache Intermediate and transparent store used to speed up future requests.

symmetric data object A local data object that has a corresponding data object on all other PEs with the same length, type and offset.

symmetric heap Special memory region, with possibly different starting address on each PE, in which dynamically created symmetric data objects are stored. Objects in this region have the same offset on every PE with respect to the region's starting address.

environment variable Set of named values, inherited from execution/launch, that will affect how programs behave.

strided-data A special type of array in which elements are separated by a specific number (non-unit) of memory locations.

E.3 Implementation Terminology

atomic operation An operation that guarantees that the resource being accessed will not be modified by another process until the operation is completed.

data latency The period of time that starts when a processing element initiates a transfer of data and ends when the processing element is able to make use of the data.

overhead	Any combination of network latency, memory bandwidth or computation time required to perform a communication operation.
mutual exclusion	Mechanism used to avoid the simultaneous use of a shared resource.
lock	Synchronization mechanism that limits access to a given resource.
undefined behavior	A behavior not defined by the OpenSHMEM specification. See the Undefined Behavior section for more information about undefined behavior in OpenSHMEM.

References

[1] ARMCI website,
http://www.emsl.pnl.gov/docs/parsoft/armci/.

[2] GASNet specification,
http://gasnet.cs.berkeley.edu/.

[3] GPSHMEM,
http://www.scl.ameslab.gov/Projects/GPSHMEM/GPSHMEM.html.

[4] PGAS forum,
http://www.pgas.org/.

[5] TurboSHMEM,
http://da.nieltiggemann.de/science/sc/turboshmem/.

[6] HPC Tools group at the University of Houston. Commonly used function calls in the OpenSHMEM library for C/C++ and FORTRAN,
http://www.openshmem.org/wiki/index.php/Documentation:Tutorials.

[7] Krzysztof Parzyszek, Ricky A. Kendall, and Robyn R. Lutz. Generalized portable SHMEM library for High Performance Computing. *Iowa State University*, 2003.

[8] Stephen W. Poole and Galen M. Shipman. Open-SHMEM: Towards a unified RMA model, 2000.

[9] Hongzhang Shan and Jaswinder Pal Singh. A Comparison of MPI, SHMEM and cache-coherent shared address space programming models on a tightly-coupled multiprocessors (sic.). *International Journal of Parallel Programming*, 29(3):283–318, June 2001.

[10] Tim Stitt. An introduction to the Partitioned Global Address Space (PGAS) programming model,
http://cnx.org/content/m20649/latest/.

[11] K. Yelick. Performance and productivity opportunities using global address space programming models, 2000.