SHMEM User's Guide for C

Ray Barriuso, Allan Knies

Revision 2.1

This document gives a user level description of the shared-memory access communication routines for the C language.

1.0 Introduction

This document presents a library to facilitate structured communications via shared memory on the Cray MPP line of computers. This new default library, named libsma.a (**S**hared **M**emory **A**ccess library), will be shipped as a part of the Programming Environment Release. The first release of this software will be in the 2nd quarter of 1994. All of the routines that are described in section 3.0 of this document will be available to the general user community at that time.

In developing this library, particular attention was paid to:

1. Only including routines/capabilities that will execute quickly (**ease of use will be sacrificed for speed and flexibility**).
2. Using a logically consistent naming technique
3. Producing a logically consistent parameterization technique
4. Assuming NUMA shared memory capabilities
5. Keeping the number of implemented operations small
6. Keeping the implementation documented, open, and expandable
7. Implementing a set of routines useful to a wide audience

In Section 2.0, we will discuss the model chosen and issues affecting development. The complete list of routines and their semantics are provided in Section 3.0. Information about future directions is provided in section 4.0.

The Fortran interface to the shmem routines can be found in [1].

2.0 Model of Computation

2.1 Memory and Processors

In this model, we will assume there are two kinds of memory: local shared and remote shared. Access to local shared memory is faster than that of remote shared memory. Both kinds of data might be cached, but that will vary from routine to routine. Where necessary, we will indicate if the programmer must take steps to maintain memory coherency.

All addresses passed in argument lists are assumed to be local. **This means that shared data descriptors or remote addresses cannot be used when calling shmem routines**. When accessing remote shared memory, the processor issuing the request must know the local virtual address for the specified data on the remote processor. This is easily accomplished by defining external data. On the Cray T3D, such data is statically allocated at the same location on all processors.

2.2 Data Types

The routines in this library will have one of the basic 64-bit data types: double, integer, or untyped and the 32-bit data types short or untyped. Untyped means that the routine manipulates memory without regard to the type of data actually stored in those locations. All three of the 64-bit data types are assumed to operate on 64-bit datum aligned on 64-bit boundaries. The 32-bit data types are assumed to operate on 32-bit datum aligned on 32-bit boundaries.

2.3 Calling Semantics

We define two kinds of routines: *collective* and *individual*. *Collective* routines require that all processors within the specified active set of processors must call the routine synchronously (start, stride, size). *Individual* routines are only called on one processor, regardless of how many processors are actually involved in the operation. The determination as to whether a routine is collective or individual will be based on both semantics and system limitations.

2.4 Grouping Model

Shmem routines allow only a subset of the processors to participate in a collective operation. These subsets are called *active sets*. In this model, there are no identifiers to name active sets nor do they need to be explicitly created. Instead, the programmer specifies the set of active processors as a tuple: (PE_start, logPE_stride, PE_size). PE_start specifies the lowest numbered processor which is to be active. logPE_stride indicates the log (base 2) of the distance between active processors. PE_size refers to the number of processors in an active set. For example, a triple of: (7, 1, 7), yields the set of processors [7, 9, 11, 13, 15, 17, 19].

When a programmer wants to use a collective library routine, they must call that routine exactly on the set of processors that are specified by the active set (PE_start, logPE_stride, PE_size) and the value of the arguments used in the call must be identical on each processor active in the operation.

This interface provides the means for development of asynchronous MIMD programs on CRI distributed memory platforms as there are no implicit global synchronizations in any of the routines.

In the following sections we use several pseudo-language constructs for ease of presentation.

- Although all of the examples and general descriptions are written in C. There is an equivalent Fortran interface described in [1].
- [4, 45, 12, 189] represents an array of 4 integers whose values are 4, 45, 12, 189.
- A variable whose name is a number (i.e. *five*) refers to a integer variable whose value is its name.
- All the examples in the following sections assume there are 8 processors in the machine.

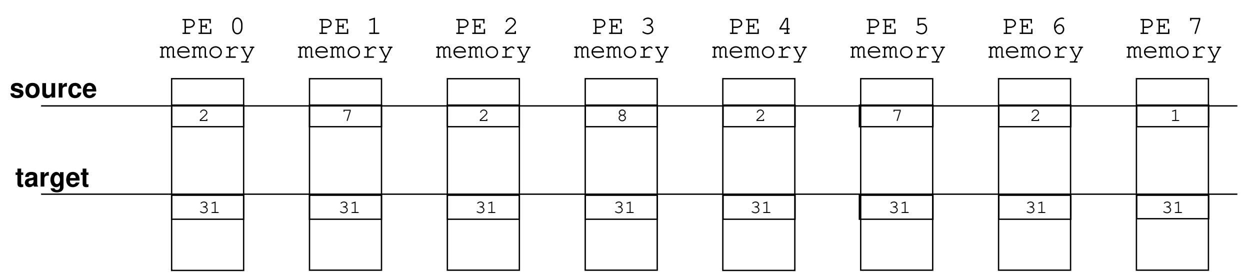
2.4.1 Simple Example

To compute an integer sum reduction[1] across eight processors, we would execute:

shmem_int_sum_to_all(target, source, 1, 0, 0, 8, pWrk, pSync);

The first two parameters indicate where the source data can be found and where the target data is to be written. The third parameter will be discussed in a later section. The fourth, fifth, and sixth parameters specify the active set as those processors starting at processor 0, at a distance of 2^0=1, for 8 processors (i.e. processors [0, 1, 2, 3, 4, 5, 6, 7]).

FIGURE 1 Picture of memory after the above example is run.



In summary:

- The programmer is responsible for making sure all the necessary processors are active for any given call to a collective subroutine.

1. A reduction operation means that a vector of operands is combined into one result. For example, a sum-reduction adds all the elements in a vector and returns the result. Some other possible reductions are: product, and, or, min, max.

- The programmer is responsible for making sure that all calling processors in a given active set are passing the same argument values including the same local virtual address for the target and source arrays.
- Any processors that are not in a particular partition are free to participate in other active sets or operations provided they do not alter memory being used by a different active set.

2.4.2 Multiple Simultaneous Independent Active Sets (same operation)

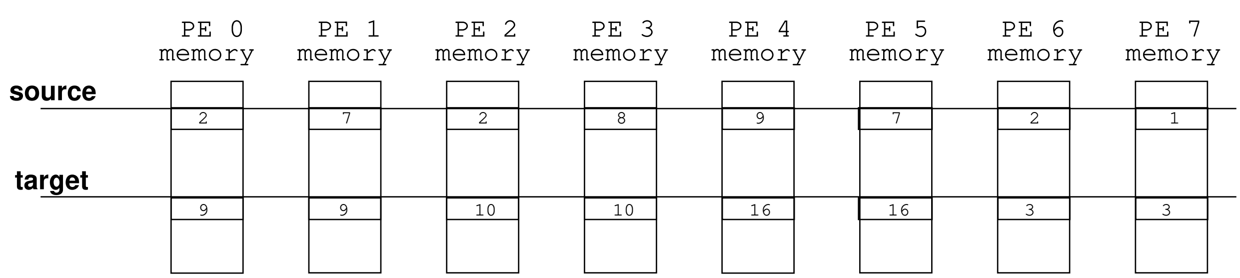
To show how multiple independent active sets can be used, suppose we want to arrange a set of virtual processor numbers in the form of a 4 X 2 grid of processors and we wish to execute computations down the columns of the processor grid. On processors 0 and 1, we will store column 0, on processors 2 and 3, we will store column 1, on processors 4 and 5, we will store column 2, and on processors 6 and 7, column 3:

If we execute:

```
PE_start = (_MY_PE()/2)*2;      /* find first processor in active set */
shmem_int_sum_to_all( target, source, 1, PE_start, 0, 2, pWrk, pSync );
```

then four simultaneous and independent reductions will take place (one on each of the four independent active sets). Notice that the value of *PE_start* will be different on each of the four active sets, but are the same within that active set. Also notice the source and target addresses are the same on each partition, but this is not a requirement in general.

FIGURE 2 Picture of memory after the above example is executed.



On the other hand, we can also execute the reductions on only the even numbered columns by executing:

```
if ( (((_MY_PE()/2)*2) % 4) == 0 ) {
    PE_start = (_MY_PE()/2)*2;        /* find first processor in active set */
    shmem_int_sum_to_all( target, source, 1, PE_start, 0, 2, pWrk, pSync );
}
```

This is legal because all the processors in two of the active sets ([0,1] and [4,5]) are participating in the call. Finally, the following example:

```
if ( _MY_PE() < 5 ) {
    PE_start = (_MY_PE()/2)*2; /* find first processor in active set */
    shmem_int_sum_to_all( target, source, 1, PE_start, 0, 2, pWrk, pSync );
}
```

is incorrect because the active set of processors does not constitute a set of complete active sets (active sets [0,1] and [2,3] are OK, but processor 4 is active and processor 5 is not).

Summarizing aspects of multiple simultaneous independent active sets:

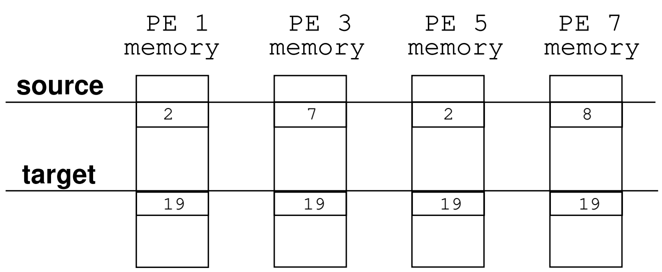
- As always, all processors in a given active set must participate in a library call and the programmer is responsible for insuring this.
- All processors in an active partition must pass the same values to the library call.

2.4.3 Non-unit striding

Users are not limited to using sets of processors that are numbered consecutively. The example below causes an integer sum reduction to be executed only on processors [1,3,5,7] (starting at processor offset 1 and striding every 2**1=2 processors, for 4 processors) and must be called synchronously on only those processors.

```
if ( _MY_PE() % 2 == 1 ) {
    shmem_int_sum_to_all( target, source, 1, 1, 1, 4, pWrk, pSync );
}
```

FIGURE 3 Picture of memory after above example is executed



Similarly:

```
if ( _MY_PE() % 4 == 0 ) {
    shmem_int_sum_to_all( target, source, 1, 0, 2, 2, pWrk, pSync );
}
```

specifies that processors [0,4] (starting at processor offset 0 and striding every 2**2 processors for two processors) are active and the reduction is to take place only on those processors.

Summarizing non-unit striding:

- The call should only be made on processors from a particular active set

- The stride is passed as the log2 of the stride desired

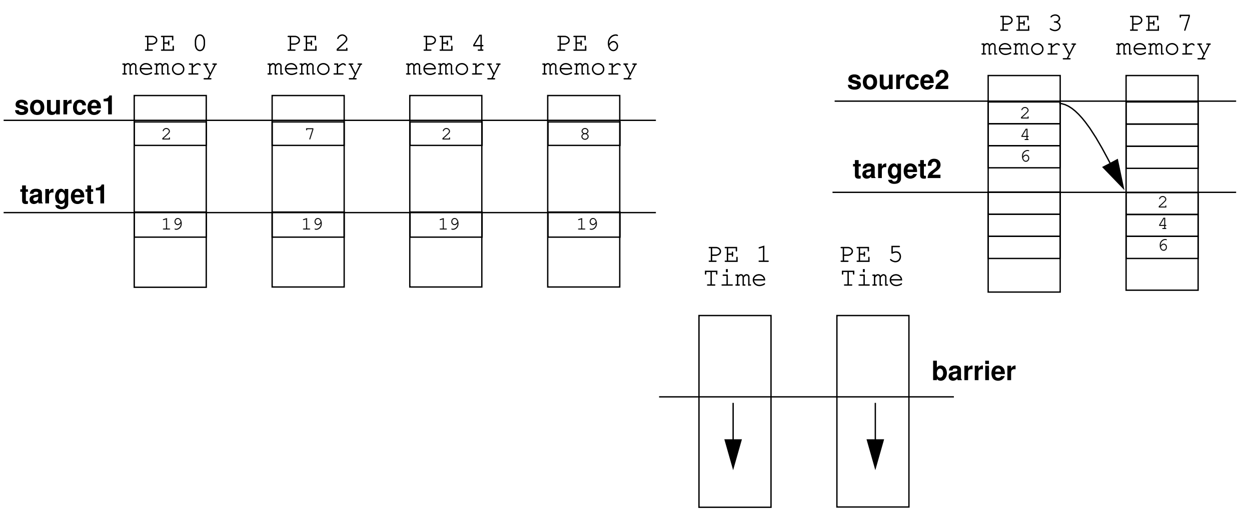
2.4.4 Multiple Simultaneous Independent Active Sets (different operations)

The examples thus far have shown how the same operation (shmem_int_sum_to_all) could be called simultaneously on independent groups of processors. Using this library, it is also possible to call different operations simultaneously provided they are called on non-overlapping sets of active processors. For example:

```
if ( _MY_PE() %2 == 0 ) { /* pes 0, 2, 4, 6 */
    shmem_int_sum_to_all( target1, source1, 1, 0, 1, 4, pWrk, pSync );
} else if ( _MY_PE() % 4 == 3 ) { /* pes 3, 7 */
    shmem_broadcast( target2, source2, 3, 0, 3, 2, 2, pSync );
} else { /* pes 1, 5 */
    shmem_barrier( 1, 2, 2, pSync );
}
```

half of the processors (processors[0,2,4,6]) would be executing an integer sum reduction, a quarter (processors [3,7]) would be executing a 3-word broadcast (from 3 to 7), and a quarter (processors [1,5]) would be synchronizing with each other.

FIGURE 4 Memory after the above code is executed.



Summarizing:

- A processor cannot be in more than one active set simultaneously.
- Multiple operations can be executed provided they operate on disjoint sets of processors.

2.5 Multiple Argument Operations

In many cases, a programmer will have a series of calls to the same library routine on the same set of processors, but with different data. To help optimize this case, many of the operations in this library can accept a count argument which indicates that the called routine needs to be executed on one or more sets of data. For example, given the following:

int source[3]; /* one three-element array in shared memory on each PE */

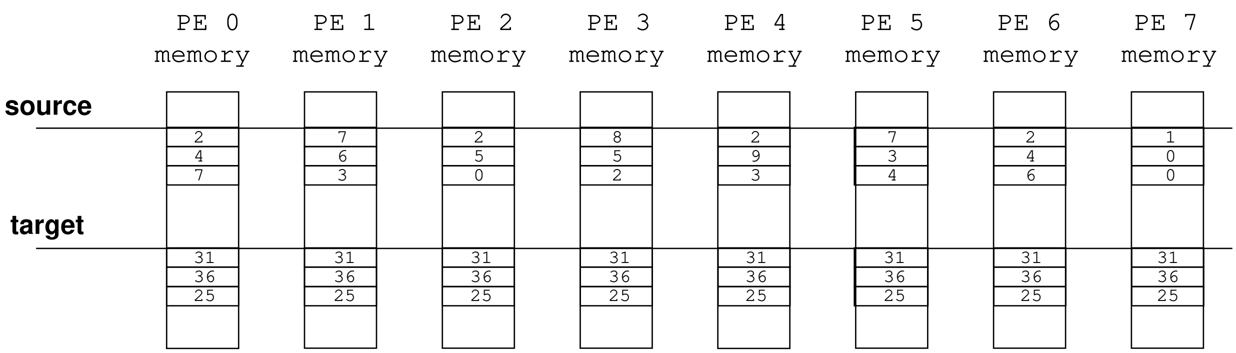
The programmer might be inclined to write:

shmem_int_sum_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8, pWrk1, pSync1);
shmem_int_sum_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8, pWrk2, pSync2);
shmem_int_sum_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8, pWrk3, pSync3);

but could more efficiently written:

shmem_int_sum_to_all(target, source, 3, 0, 0, 8, pWrk, pSync);

FIGURE 5 shmem_int_sum_to_all(target, source, 3, 0, 0, 8, pWrk, pSync). Notice which processors are involved in the reduction and where source and result (target) data are placed.



Summarizing:

- In all cases, if the parameter specifying operation count is greater than one, then the data must be in consecutive memory locations starting at the address passed as the *source* operand and the results are written into consecutive memory locations starting at the *target* address.

3.0 Usage Conventions/General Semantics/Routine Descriptions

In this section, we will describe and document routines based on the discussion above. Some routines may have system specific notes.

3.1 Format of Collective Routines

In general, the format of the parameter lists for collective routines is:

```
OpName( target, source, 3, 0, 0, 2, pWrk, pSync )
```

Memory Address Information (target, source) — Multiple Issue Information (3) — Active SET Information (0, 0, 2) — Workspace (pWrk, pSync)

3.2 Important Usage Issues

- **In general, to insure cache coherency the remote write cache invalidate bit should be set in all processors prior to any library routines being called.** This bit can be set by calling *shmem_set_cache_inv()* and cleared by calling *shmem_clear_cache_inv()*. The knowledgable, careful user can alternatively flush the entire cache or a particular cache line at key points in the program by calling *shmem_udcflush()* or *shmem_udcflush_line()* respectively. This is a T3D specific issue. Some routines may relax this constraint.
- The *logPE_stride* argument is always the $\log_2$ of the stride required (i.e. to get a stride of 4, the programmer would pass in a value of 2).
- PE_size must be less than or equal to the number of available processors.
- Target and source operands must not point to partially overlapping memory regions unless otherwise noted. That is, they must point to the same region in memory or to completely disjoint regions in memory.
- **When a collective routine returns, it does not imply that the operation has completed on remote processors**. Thus, before any results on remote processors can be accessed, the programmer must be sure that the operation has completed (via a barrier or some other synchronization technique).
- Many routines require user allocated work space. **Workspace must be allocated at the same address in each processor (that is, allocated in static memory).**

  Generally, there are two kinds of workspace required: data and sync. Data workspace need not be initialized. The sync workspace must be initialized to _SHMEM_SYNC_VALUE (defined in <mpp/shmem.h>) before any processor tries to access it (i.e. user must guarantee no one will access sync workspace until it is initialized). **Typically, a barrier is place between the initialization of the sync workspace array and the first call to a collective routine.** For example:

```
for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
      pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier();      /* wait for all processor to initialize pSync */
      :
shmem_broadcast(tar, src, size, root, start, stride, npes, pSync);
```

3.3 Performance Issues

- Operands for routines in the library will generally run faster if the arguments passed to it are cache-aligned (in some cases, not much, in others, quite a bit). Try to align data along cache-lines wherever possible.
- Although the partitioning model allows strange shaped subsets to be used, the routines are optimized for a 3D rectangular torus whose extents are powers of two.

Barrier

#include <mpp/shmem.h>
void shmem_barrier(**int** *PE_start*, **int** *logPE_stride*, **int** *PE_size*, **long** **pSync*);

Types N/A

Calling Semantics Collective

Multiple Arguments N/A

Description This routine causes a barrier operation over the set of processors specified by *(PE_size, PE_start, logPE_stride)*.

Usage Issue The programmer must allocate and initialize work space with enough room for _SHMEM_BARRIER_SYNC_SIZE integers and pass a pointer to that space via the *pSync* argument. Each integer in the buffer must be initialized to _SHMEM_SYNC_VALUE (defined in <mpp/shmem.h>) by all processors before *shmem_barrier()* is called the first time by any processor. For example:

```
for (i=0; i < _SHMEM_BARRIER_SYNC_SIZE; i++) {
      pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier();      /* wait for all processor to initialize pSync */
       :
shmem_barrier(start, stride, npes, pSync);
```

The programmer need only initialize the buffer the first time it is used as *shmem_barrier*() resets the contents as it needs, for the next call.

Usage Issue The programmer can only reuse a buffer if it is known that no other active routine might use it before all processors have passed the barrier. In the most general case, this can only be known by inserting a call to a barrier routine.

However, in the case where a barrier is required on the same partition of processors multiple times, the same buffers can be used without intervening synchronization points.

Broadcast

#include <mpp/shmem.h>
void shmem_broadcast(**long** **target*, **long** **source*, **int** *nlong*, **int** *PE_root*, **int** *PE_start*, **int** *logPE_stride*, **int** *PE_size*, **long** **pSync*);

Types Untyped

Calling Semantics Collective

Multiple Arguments No

Description This routine broadcasts a block of data of length *nlong* 64-bit words, starting at local address *source* on the processor specified by *PE_root*, and stores the values at *target* on each processor specified by (*PE_size, PE_start, logPE_stride*). The data is not copied to the target area on the root PE.

Remember, every active processor must call *shmem_broadcast()* with the same values passed to each argument.

PE_root specifies the number within the active set to be the root. That is, to broadcast from the first processor in the active set, *PE_root* should be zero regardless of what processors are active in the active set.

Usage Issue The *source* and *target* parameters must be aligned on a 64-bit boundary

Usage Issue The programmer must allocate and initialize a synchronization buffer on each processor that is large enough to hold _SHMEM_BCAST_SYNC_SIZE integers and pass a pointer to it via the *pSync* argument. Each element of the buffer must be initialized to _SHMEM_SYNC_VALUE by all processors before calling *shmem_broadcast()* the first time by any processor. For example:

```
for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
      pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier();       /* wait for all processor to initialize pSync */
      :
shmem_broadcast(tar, src, size, root, start, stride, npes, pSync);
```

shmem_broadcast() automatically resets the local buffer values before returning so the buffer need only be initialized before the first use.

Usage Issue The programmer can only reuse a buffer if it is known that no other active routine might use it before all processors have finished the broadcast. In general, this can only be known by passing a barrier or other synchronization point.

Cache Management

void shmem_set_cache_inv(**void**);
void shmem_set_cache_line_inv(**long** **target*);
void shmem_clear_cache_inv(**void**);
void shmem_udcflush(**void**);
void shmem_udcflush_line(**long** **target*);

Types	Untyped
Calling Semantics	Individual
Multiple Arguments	No
Description	*shmem_set_cache_inv()* and *shmem_clear_cache_inv()* enable and disabled automatic cache invalidation for all writes into local memory by other processors. *shmem_set_cache_line_inv()* enables automatic cache invalidation for only the cache line associated with the specified address. *shmem_udcflush()* invalidates the entire user data cache. Current timings indicate that invalidating the data cache with this routine takes approximately 230 cps. *shmem_udcflush_line()* invalidates the cache line that corresponds with the address specified by target. Current timings indicate that invalidating a cache line with this routine takes approximately 27 cps.
Usage Issue	These routines are Cray T3D specific and will most likely not exist in libraries supporting future Cray MPP architectures.

Collect

#include <mpp/shmem.h>
void shmem_collect(**long** **target*, **long** **source*, **int** *nwords*, **int** *PE_start*, **int** *logPE_stride*, **int** *PE_size*, **long** **pSync*)
void shmem_fcollect(**long** **target*, **long** **source*, **int** *nwords*, **int** *PE_start*, **int** *logPE_stride*, **int** *PE_size*, **long** **pSync*)

Types Untyped

Calling Semantics Collective

Multiple Arguments Yes

Description These routines collect and concatenate *nwords* of 64-bit data from address *source* into address *target* over the set of processors defined by *(PE_size, PE_start,* log_2*PE_stride)* in processor number order. That is, the resultant *target* array contains the contribution from processor *PE_start* first, then the contribution from processor *PE_start* + *PE_stride* second, and so on. The collected result is written to all active processors at address *target*. Routine *shmem_collect()* allows the value of *nwords* to be different on each participating processor. Routine *shmem_fcollect()*, however, expects the value of *nwords* to be the same for all processors. Since the value for *nwords* is the same on all processors for calls to *shmem_fcollect()*, the speed of this routine is greatly improved over routine *shmem_collect()* due to the reduced overhead of calculating where each contribution belongs in the target array. These routines will operate on active set sizes that are not powers of two (with some performance degradation) and for values of *nwords* that are not powers of two (no performance degradation).

Usage Issue The programmer must supply a synchronization buffer, *pSync*. *pSync* must be large enough to hold at least $2 \times \lceil \log PEsize \rceil + 2$ integers.

pSync must have all its values initialized to _SHMEM_SYNC_VALUE before the first time it used for a given active set. For example:

```
for (i=0; i < SYNC_SIZE; i++) {
      pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier();       /* wait for all processors to initialize pSync */
      :
shmem_fcollect(tar, src, 64, start, stride, npes,  pSync);
```

The collect routines reset the values in *pSync* before they return to the caller so a particular sync buffer need only be initialized the first time it is used.

Usage Issue *pSync* may only be reused if making calls on active sets that do not contain overlapping processors or if a synchronization occurs in between calls to the collect routines.

However, in the special case where many collections are to be performed on the same active set without wanting to synchronize, the programmer can allocate two buffers and alternate between them on successive calls.

Get/Put

void shmem_get(**long** **target*, **long** **source*, **int** *nlong*, **int** *pe*);
void shmem_put(**long** **target*, **long** **source*, **int** *nlong*, **int** *pe*);

void shmem_get32(**short** **target*, **short** **source*, **int** *nlong*, **int** *pe*);
void shmem_put32(**short** **target*, **short** **source*, **int** *nlong*, **int** *pe*);

Types Untyped

Calling Semantics Individual

Multiple Arguments No

Description In the following description, statements concerning *shmem_get()* and *shmem_put()* apply equally to *shmem_get32()* and *shmem_put32()* unless noted otherwise.

These routines copy blocks of data directly from one processor's memory to another. The *nlong* parameter indicates the number of 64-bit words (for *shmem_put()* and *shmem_get()*) or 32-bit words (for *shmem_put32()* and *shmem_get32()*) to be transferred. For *shmem_get()* datum is copied starting at address *source* on the processor specified by *pe* and placed at address *target* in the calling processor's memory. For *shmem_put()* datum is copied starting at address *source* in the calling processor's memory and placed at address *target* on the processor specified by *pe*.

Usage Issue Since *shmem_put()* writes directly into another processor's memory without that processor being notified, the remote write invalidate bit should be set on the receiving processor before any processor writes into its memory, otherwise, the cache on the remote processor may become incoherent.

The programmer can use *shmem_put()* without setting the remote write invalidate bit if they know prior to the call that the data at the target addresses are not cachable or otherwise known not to reside in the remote processor's cache at the time of the call. However, it is very easy for a programmer to make a mistake trying to do this and thus it is highly recommended that the remote write invalidate bit be set on (unfiltered mode) in any processor whose memory is being updated remotely. This bit is set by calling either *shmem_set_cache_inv()* or *shmem_set_cache_line_inv()* and cleared by calling *shmem_disable_cache_inv()*.

It is also possible for the receiving processor to flush its cache when it knows that it has received data from another processor. In general, this will require synchronization. The cache can be flushed using the routines *shmem_udcflush()* or *shmem_udcflush_line()*.

Usage Issue *shmem_put()* returns as soon as the CPU is done sending the data to the network. Thus, even though the routine has returned, the data may not yet have been stored on the remote processor. To guarantee this, the user can call the MPP C compiler intrinsic *_remote_write_barrier()*. *_remote _write_barrier()* blocks a processor until all of its remote writes have reached their destination.

Usage Issue *shmem_put()* reads from local memory addresses. If local memory is not coherent before the call, it may read stale data from the local cache.

Usage Issue *shmem_get()* always reads from memory even if the cache is incoherent.

Strided Get/Put

void shmem_iget(**long** **target*, **long** **source*, **int** *target_inc*, **int** *source_inc*, **int** *nlong*, **int** *pe*);
void shmem_iput(**long** **target*, **long** **source*, **int** *target_inc*, **int** *source_inc*, **int** *nlong*, **int** *pe*);

Types Untyped

Calling Semantics Individual

Multiple Arguments No

Description These routines are extensions to the basic *shmem_get()* and *shmem_put()* routines. *shmem_iget()* allows strided single word loads from another processor's memory. This is semantically equivalent to:

```
for ( i=0; i<nlong; i++ ) {
    shmem_get( target, source, 1, pe );
    source += source_inc;
    target += target_inc;
}
```

shmem_iput() is similar except that it allows strided single word stores to another processor's memory instead of loads.

Usage Issue All usage issues for *shmem_put()* and *shmem_get()* apply to *shmem_iput()* and *shmem_iget()* as well.

Gather/Scatter

void shmem_ixget(**long** **target*, **long** **source*, **long** **source_index*, **int** *nlong*, **int** *pe*);
void shmem_ixput(**long** **target*, **long** **source*, **long** **target_index*, **int** *nlong*, **int** *pe*);

Types Untyped

Calling Semantics Individual

Multiple Arguments No

Description These routines are extensions to the basic *shmem_get()* and *shmem_put()* routines. *shmem_ixget()* implements a point-to-point gather operation from another processor's memory. This is semantically equivalent to:

```
for ( i=0; i<nlong; i++ ) {
    shmem_get( &(target[i]), source + source_index[i], 1, pe );
}
```

shmem_ixput() is similar except that it implements a point-to-point scatter operation to another processor's memory.

Usage Issue All usage issues for *shmem_put()* and *shmem_get()* apply to *shmem_ixput()* and *shmem_ixget()* as well.

Reduction

#include <mpp/shmem.h>

void shmem_X_Y_to_all(**X** ******target*, **X** ******source*, **int** *nreduce*, **int** *PE_start*, **int** *logPE_stride*,
int *PE_size*, **X** ******pWrk*, **long** **pSync*)

Types X = (int, double)

Y = (sum, prod, min, max, and, or)[1]

Calling Semantics Collective

Multiple Arguments Yes

Description These routines compute a reduction Y on data type X using address *source* over the set of processors defined by *(PE_size, PE_start, logPE_stride)*. By specifying *nreduce* > 1, this routine will compute multiple independent reductions. The results from the reduction are written to all active processors at address *target*. This routine will operate on active set sizes that are not powers of two (with some performance degradation) and for values of *nreduce* that are not powers of two (no performance degradation).

Usage Issue *source* and *target* must not point to partially overlapping regions in memory, but may point to the same or disjoint regions.

Usage Issue The programmer must supply two workspace buffers. For optimal performance, these work space buffers should be cache aligned.

pWrk is data workspace and must be large enough to hold at least max[(*nreduce*/2) +1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE] values of type X. The size of the buffer should be a multiple of the cache line size (4 for the T3D).

pSync is synchronization workspace and must be large enough to hold at least $2 \times \lceil \log PEsize \rceil + 2$ integers. The size of the buffer should be a multiple of the cache line size.

pSync must have all its values initialized to _SHMEM_SYNC_VALUE before the first time it used for a given active set. For example:

```
for (i=0; i < SYNC_SIZE; i++) {
      pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier();      /* wait for all processor to initialize pSync */
      :
shmem_int_max_to_all(tar, src, size, start, stride, npes, pWrk, pSync);
```

shmem_X_Y_to_all() resets the values in *pSync* before it returns so a particular sync buffer need only be initialized the first time it is used.

1. Operations *and* and *or* are not supported for double/REAL8 types

Usage Issue

Workspace may only be reused if making calls on active sets that do not contain overlapping processors or if a synchronization occurs in between calls to the reduction routines.

However, in the special case where many reductions are to be performed on the same active set without wanting to synchronize, the programmer can allocate two buffers and alternate between them on successive calls.

Usage Issue

source and *target* data must be left intact until it is known that all other processors are done referencing that information. In general, this requires a synchronization point.

Wait

void shmem_wait(**long** **paddr*, **int** *flag_value*)

Calling Semantics Individual

Multiple Arguments n/a

Description *shmem_wait()* is used for point to point directed synchronization. When a processor calls *shmem_wait()*, the call will not return until some other processor writes a value not equal to *flag_value* into the waiting processor's memory at address *paddr*. This mechanism is useful when a processor needs to tell another processor that it has completed some action.

Implementation This routine invalidates the cache line associated with address *paddr*, updates the cache invalidation filter so that remote writes to this location will invalidate the corresponding cache line, and then waits by looping in cache until the value at address *paddr* does not equal the value specified by *flag_value*.

Note that at times it may be more efficient to write a simple spin-wait loop with a variable that is declared volatile. This will be the case if other processors will not be writing blocks of data into the waiting processor. It takes *shmem_wait()* about 100 clock periods to set up the cache invalidation filter and another 100 clock periods to disable the filter before returning to the caller.

Usage Issue When a processor waits on an address, the programmer must be sure that the signalling processor writes to the correct location. Generally, this is done by declaring a 'flag' variable at the same address on each processor and using the address of the local flag to signal the remote processor.

Usage Issue A brief coding example for *shmem_wait()* follows:

```
. . .
flag_var = flag_value;    /* initialize the event variable */
. . .
if (flag_var == flag_value) {
      shmem_wait( &flag_var, flag_value );
}
flag_var = flag_value;     /* reset the event variable for next time */
. . .
```

4.0 Future Directions

As the library is currently constructed, it has a fairly small base of functionality. There may be room to add more functionality. The following routines may be useful, but have not yet been shown to be sufficiently fast or sufficiently general to be included (or we haven't had time to implement them).

This library seeks performance and simplicity. The general philosophy is that a new routine should not be added unless it can be implemented significantly faster than a good high-level language programmer can using tools that are already available in the libsma.a library or from high level language constructs.

All new collective routines will support the (start, stride, size) partitioning model and will keep the parameter lists in the same order as shown in this document.

Several of the routines below are extensions to the basic shmem_get() routine. There are analogous versions of shmem_put() that could be included as well.

4.1 shmem_get_X(target, local_source, remote_source, nlong, pe)

This would be a family of routines where X represents a binary operator. This function would be semantically equivalent to:

```
shmem_get(tmp_v, remote_source, nlong, pe);
for ( i=0; i<nlong; i++ ) {
      target[i] = tmp_v[i] + local_source[i];
}
```

This would allow values fetched remotely to be incorporated directly into a computation rather than re-reading it as in the code shown above. Note that cache conflicts could cause this routine to be very slow if the implementation is not careful!

4.2 Multi-PE scatter/gather/indexed/strided

Similar to i and ix variants of shmem_put() and shmem_get(), but across processors instead.

4.3 prefix

This routine is the binary all-partial-sums routine. Its parameter list would look very similar to shmem_x_y_to_all().

4.4 reduce_to_one()

Version of shmem_x_y_to_all that only sends its result to one processor instead of to all.

References

[1] R. Barriuso, "SHMEM User's Guide for Fortran," Technical Report, Cray Research Inc., Eagan, MN., June 20, 1994.