NAME

intro_shmem – Introduction to logically shared memory access routines

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The logically shared, distributed memory access (SHMEM) routines perform various operations on remote and local memory. These routines can minimize overhead and latency and can maximize data bandwidth. Supported operations include remote data transfer, atomic swap, work-shared broadcast and reduction, and barrier synchronization.

The SHMEM routines are described on the following man pages:

- num_pes(3)
- shmem_and(3)
- shmem_barrier(3)
- shmem_broadcast(3)
- shmem_cache(3)
- shmem_collect(3)
- shmem_iput(3)
- shmem_ixput(3)
- shmem_lock(3)
- shmem_max(3)
- shmem_min(3)
- shmem_my_pe(3)
- shmem_or(3)
- shmem_prod(3)
- shmem_put(3)
- shmem_quiet(3)
- shmem_stack(3)
- shmem_sum(3)
- shmem_swap(3)
- shmem_wait(3)
- shmem_xor(3)
- start_pes(3)

C or Fortran data objects are passed by address to SHMEM routines. Target or source arrays that reside on the remote processing element (PE) are typically identified by passing the address of the corresponding data object on the local PE. The existence of a corresponding data object locally implies that a data object is *symmetric* as described on this man page. Most uses of SHMEM routines employ symmetric data objects, but on Cray MPP systems, another class of data object, *asymmetric accessible* data objects, can also be passed to SHMEM routines.

Data objects passed to SHMEM routines can be arrays or scalars, and they fall into the following addressing categories:

Category	Characteristics
Symmetric	A remotely accessible data object whose local address maps in a consistent way to a corresponding data object on remote PEs. Remote data objects with this property are accessed through SHMEM routines by using the address of the corresponding data object on the local PE. All symmetric data objects are also *PE-private*. The following data objects are symmetric on Cray MPP systems: • Fortran PE-private data objects in common blocks or with the `SAVE` attribute • Fortran arrays allocated with `shpalloc`(3F) • C variables with static storage class • C data allocated by `shmalloc`(3C) • C stack variables declared with a `#pragma symmetric` directive • Fortran stack variables declared with a `CDIR$ SYMMETRIC` directive (CF90 compiler only) The following data objects are symmetric on Cray PVP systems: • Fortran arrays in `TASK COMMON` • C arrays declared in `#pragma taskcommon` directives
Asymmetric accessible	A PE-private data object with a virtual address that can be accessed from other PEs through SHMEM routines. Asymmetric accessible data objects exist on Cray MPP systems only. A data object that is remotely accessible, but not symmetric, can be accessed on Cray MPP systems through SHMEM routines only if the address of the remote data object is communicated between PEs first. The address of the data object on the remote PE must be passed to the SHMEM routines in those cases. All asymmetric accessible data objects are also PE-private (see below). The following data objects are asymmetric accessible on Cray MPP systems: • C data allocated by `malloc`(3C), if the same address is a valid heap address on all other PEs • C variables with automatic storage class, if the private stacks on all other PEs have been extended to the same virtual address • Fortran arrays allocated with `hpalloc`(3F), if the same address is a valid heap address on all other PEs • Fortran PE-private data objects on the stack, if the private stacks on all other PEs have been extended to the same virtual address
Remotely accessible	Any symmetric or asymmetric accessible data object.
PE-private	Any data object that is not an MPP Fortran shared array. This includes all C and Fortran data objects not declared `SHARED`.

These classifications divide data objects according to the address categories. These classifications apply to all programs except Cray MPP programs linked with the `-O SHARED` option on `mppldr`(1). For those programs, only Fortran arrays allocated with `shpalloc`(3F) and C data allocated by `shmalloc`(3C) are symmetric. All other data objects are not symmetric or asymmetric accessible.

SHMEM routines that operate on the same data object on multiple PEs require that symmetric data objects be passed. This restriction is for algorithm simplicity and efficiency. These routines define the set of target PEs by the following triplet of arguments: *PE_start*, *logPE_stride*, and *PE_size*.

Some SHMEM routines, for example, `shmem_broadcast`(3) and `shmem_float_sum_to_all`(3), are classified as *collective* routines because they distribute work across a set of PEs. They must be called concurrently by all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* argument triplet.

ROUTINES ON CRAY PVP SYSTEMS

A small subset of the SHMEM routines are available on Cray PVP systems. The IMPLEMENTATION section of each man page indicates if a routine is available on Cray PVP systems. A summary of routines available on Cray PVP systems follows.

The following routines are available from both Fortran and C on Cray PVP systems:

- `barrier`(3C)
- `barrier`(3F)
- `shmem_clear_cache_inv`(3)
- `shmem_get`(3)
- `shmem_my_pe`(3)
- `shmem_n_pes`(3)
- `start_pes`(3)
- `shmem_put`(3)
- `shmem_quiet`(3)
- `shmem_set_cache_inv`(3) (unavailable on CRAY T90 systems)
- `shmem_set_cache_line_inv`(3) (unavailable on CRAY T90 systems)
- `shmem_udcflush`(3)
- `shmem_udcflush_line`(3)

The following routine is available from Fortran only (this routine is not intrinsic on Cray PVP systems):

- `MY_PE`(3I)

The following routines are available from C only (neither of these are intrinsic on Cray PVP systems):

- `_my_pe`(3)
- `_num_pes`(3)

REDUCTION ROUTINES

The SHMEM reduction routines perform reductions on symmetric arrays. They are described on the following man pages:

- `shmem_and`(3)
- `shmem_max`(3)
- `shmem_min`(3)
- `shmem_or`(3)
- `shmem_prod`(3)
- `shmem_sum`(3)
- `shmem_xor`(3)

COLLECTIVE ROUTINES

The following man pages describe SHMEM collective routines:

- shmem_and(3)
- shmem_barrier(3)
- shmem_broadcast(3)
- shmem_collect(3)
- shmem_max(3)
- shmem_min(3)
- shmem_or(3)
- shmem_prod(3)
- shmem_sum(3)
- shmem_xor(3)

USING THE *pSync* ARRAY

Multiple *pSync* arrays are often needed if a particular PE calls a SHMEM routine twice. Problems would occur if some PEs in the active set for call 2 arrive at call 2 before processing of call 1 is complete by all PEs in the call 1 active set. Another way to prevent call 2 from beginning before barrier processing at call 1 is complete is to call barrier(3C) or barrier(3F) between call 1 and call 2.

There are two special cases:

- The shmem_barrier(3) routine allows the same *pSync* array to be used on consecutive calls as long as the active PE set does not change.
- If the same collective routine is called multiple times with the same active set, the calls may alternate between two *pSync* arrays. The SHMEM routines guarantee that a first call is completely finished by all PEs by the time processing of a third call begins on any PE.

Because the SHMEM routines restore *pSync* to its original contents, multiple calls that use the same *pSync* array do not require that *pSync* be reinitialized after the first call.

MISCELLANEOUS

Note that Fortran shared variables cannot be passed to any of the SHMEM routines.

Performance of some SHMEM routines is enhanced by using data objects that are cache aligned.

The SHMEM routines reside in libsma.a, which is a library in the default mppldr(1) search path on Cray MPP systems.

The Fortran SHMEM header file for CRAY T3D systems resides in directory /usr/include/mpp on Cray PVP systems. Use the -I /usr/include/mpp option with the f90(1) or cf77(1) commands when compiling Fortran modules that include a SHMEM Fortran header file for CRAY T3D systems.

SEE ALSO

num_pes(3), shmem_and(3), shmem_barrier(3), shmem_broadcast(3), shmem_cache(3), shmem_collect(3), shmem_iput(3), shmem_ixput(3), shmem_lock(3), shmem_max(3), shmem_min(3), shmem_my_pe(3), shmem_or(3), shmem_prod(3), shmem_put(3), shmem_quiet(3), shmem_stack(3), shmem_sum(3), shmem_swap(3), shmem_wait(3), shmem_xor(3), start_pes(3)

cf77(1), f90(1)

mppldr(1)

barrier(3C), shmalloc(3C)

barrier(3F), BARSYNC(3F), hpalloc(3F), shmalloc(3F), shpalloc(3F)

my_pe(3I)

PVM and HeNCE Programmer's Manual, publication SR–2501

CRAY MPP Fortran Reference Manual, publication SG–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

num_pes – Returns the number of processing elements (PEs) running in an application

SYNOPSIS

C:

```
int  num_pes(void);
```

Fortran:

```
I  =  NUM_PES()
```

IMPLEMENTATION

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

This function returns the number of PEs running in an application. This is the number of PEs started with the start_pes(3) routine.

SEE ALSO

intro_shmem(3), my_pe(3I)

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_short_and_to_all, shmem_int_and_to_all, shmem_int8_and_to_all, shmem_int4_and_to_all – Performs a logical AND function across a set of processing elements (PEs)

SYNOPSIS

C:

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

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);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

CALL SHMEM_INT8_AND_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the result of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* date types are as follows:

Routine	Data Type
`shmem_int8_and_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_and_to_all`	Integer, with an element size of 4 bytes

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in intro_shmem(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the logical AND of the integer variable FOO across all even PEs.

```
INCLUDE "mpp/shmem.fh"

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

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
   CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, N$PES/2,
&      PWRK, PSYNC)
   PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOAND
ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_and_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_and_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_and_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_and_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

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

SYNOPSIS

C:

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

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

Fortran:

```
INCLUDE "mpp/shmem.fh"

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

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

shmem_barrier is a collective synchronization routine. Control returns from shmem_barrier after all PEs in the active set (specified by *PE_start*, *logPE_stride*, and *PE_size*) have called shmem_barrier.

The arguments are as follows:

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pSync A symmetric work array.

In C, *pSync* must be of type int and size _SHMEM_BARRIER_SYNC_SIZE.

In Fortran, *pSync* must be of type integer and size SHMEM_BARRIER_SYNC_SIZE. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE (in C) or SHMEM_SYNC_VALUE (in Fortran) before any of the PEs in the active set enter shmem_barrier.

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

shmem_barrier ensures that all previously issued local and remote memory stores by any of the PEs in the active set are complete before returning. The data cache is not invalidated, however.

shmem_barrier restores the contents of the *pSync* array to the original values on all PEs before any PE returns. This implies that the same *pSync* array may be used on consecutive calls to shmem_barrier if the same active PE set is used.

NOTES

The term *symmetric* is defined in intro_shmem(3).

shmem_barrier resets the values in *pSync* before it returns, so a particular *pSync* buffer needs to be initialized only the first time it is used. If the *pSync* array is initialized at run time, be sure to use some type of synchronization, for example, a call to barrier(3F) or barrier(3C), before calling shmem_barrier 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.

EXAMPLES

C example:

```
for (i=0; i < _SHMEM_BARRIER_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier(); /* Wait for all PEs to initialize pSync */
shmem_barrier(PE_start, logPE_stride, size, pSync);
```

Fortran example:

```
INTEGER PSYNC(SHMEM_BARRIER_SYNC_SIZE)
INTEGER PE_START, LOGPE_STRIDE, PE_SIZE, PSYNC
DATA PSYNC /SHMEM_BARRIER_SYNC_SIZE*SHMEM_SYNC_VALUE/

CALL SHMEM_BARRIER(PE_START, LOGPE_STRIDE, PE_SIZE, PSYNC)
```

SEE ALSO

barrier(3C), barrier(3F), intro_shmem(3), set_barrier(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_broadcast, shmem_broadcast32, shmem_broadcast4 – Broadcasts a block of data from one processing element (PE) to one or more target PEs

SYNOPSIS

C:

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

void shmem_broadcast(void *target, void *source, int nlong,
   int PE_root, int PE_start, int logPE_stride, int PE_size, long *pSync);

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

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong, PE_root, PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
CALL SHMEM_BROADCAST(target, source, nlong, PE_root, PE_start,
   logPE_stride, PE_size, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

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

The arguments are as follows:

target A symmetric data object with one of the following data types:

Routine	Data Type and Language
shmem_broadcast	Any noncharacter type that has an element size of 64 bits. No Fortran derived types or C structures are allowed.
shmem_broadcast32	Any noncharacter type that has an element size of 32 bits. No Fortran derived types or C structures are allowed.
shmem_broadcast4	Any noncharacter type that has an element size of 32 bits.

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

nlong The number of elements in *source*. For `shmem_broadcast`, this is the number of 64-bit words. For `shmem_broadcast32` and `shmem_broadcast4`, this is the number of 32-bit halfwords. *nlong* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_root Zero-based ordinal of the PE in the active set from which the data is copied. Must be greater than or equal to 0 and less than `PE_size`. *PE_root* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *log_PE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_BCAST_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_BCAST_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter `shmem_barrier`().

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

Upon return from a broadcast routine, the following are true for the local PE:

- If the current PE is not the root PE, the *target* data object is updated.
- The data cache region that is mapped to the *target* data object is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM broadcast routines restore *pSync* to its original contents. Multiple calls to SHMEM routines that use the same *pSync* array do not require that *pSync* be reinitialized after the first call.

You must ensure the that the *pSync* array is not being updated by any PE in the active set while any of the PEs participates in processing of a SHMEM 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 a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* array may be reused on a subsequent SHMEM broadcast routine only if none of the PEs in the active set are still processing a prior SHMEM 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 SHMEM routines being called with the same active set, you can allocate two *pSync* arrays and alternate between them on successive calls.

EXAMPLES

C example:

```
for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier(); /* Wait for all PEs to initialize pSync */
shmem_broadcast(target, source, nlong, PE_root, PE_start,
  logPE_stride, PE_size, pSync);
```

Fortran example:

```
      INTEGER PSYNC(SHMEM_BCAST_SYNC_SIZE)
      INTEGER TARGET, SOURCE, NLONG, PE_ROOT, PE_START,
     &      LOGPE_STRIDE, PE_SIZE, PSYNC
      COMMON /COM/ TARGET, SOURCE

      DATA PSYNC /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/

      CALL SHMEM_BROADCAST(TARGET, SOURCE, NLONG, PE_ROOT, PE_START,
     &      LOGPE_STRIDE, PE_SIZE, PSYNC)
```

SEE ALSO

`intro_blacs`(3S), `intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_set_cache_inv, shmem_set_cache_line_inv, shmem_clear_cache_inv, shmem_udcflush, shmem_udcflush_line – Controls data cache utilities

SYNOPSIS

C:

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

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);
```

Fortran:

```
INCLUDE "mpp/shmem.fh"

CALL SHMEM_SET_CACHE_INV

CALL SHMEM_SET_CACHE_LINE_INV(target)

CALL SHMEM_CLEAR_CACHE_INV

CALL SHMEM_UDCFLUSH

CALL SHMEM_UDCFLUSH_LINE(target)
```

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

shmem_set_cache_inv and shmem_set_cache_line_inv are not supported on CRAY T90 systems

DESCRIPTION

The following argument is passed to the cache line control routines:

target A data object that is local to the processing element (PE). *target* can be of any noncharacter type. If you are using the CF90 compiler, it can be of any kind.

shmem_set_cache_inv enables automatic cache coherency mode. When this mode is enabled on CRAY T3D systems, remote writes into the calling PE's local memory from other PEs via shmem_put(3) or shmem_swap(3) do not disrupt data cache coherency because the affected cache line in the local PE gets invalidated automatically.

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

`shmem_set_cache_inv` and `shmem_set_cache_line_inv` are no-ops on CRAY Y-MP systems, CRAY J90 systems, CRAY C90 systems, and CRAY T3E systems because no data cache exists or because hardware or the operating system ensures cache coherency.

`shmem_set_cache_inv` and `shmem_set_cache_line_inv` are not provided on CRAY T90 systems because hardware automatic cache coherency mode is unavailable.

`shmem_clear_cache_inv` disables automatic cache coherency mode previously enabled by `shmem_set_cache_inv` or `shmem_set_cache_line_inv`.

On CRAY Y-MP systems, CRAY J90 systems, CRAY C90 systems, and CRAY T3E systems, automatic cache coherency cannot be disabled, so `shmem_clear_cache_inv` is a no-op. On CRAY T90 systems, automatic cache coherency mode is unavailable, so `shmem_clear_cache_inv` is also a no-op.

`shmem_udcflush` makes the entire user data cache coherent.

On CRAY Y-MP systems, CRAY J90 systems, CRAY C90 systems, and CRAY T3E systems, `shmem_udcflush` and `shmem_udcflush_line` are no-ops because no data cache exists or because hardware or the operating system ensures cache coherency.

On CRAY T3D systems and CRAY T90 systems, `shmem_udcflush` makes the data cache coherent by invalidating it. This has the same effect as the `#pragma _CRI suppress` directive (in C) and the `CDIR$ SUPPRESS` directive (in Fortran).

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

On CRAY Y-MP systems, CRAY J90 systems, CRAY C90 systems, and CRAY T3E systems, `shmem_udcflush_line` is a no-op because no data cache exists or because hardware or the operating system ensures cache coherency.

On CRAY T3D systems, `shmem_udcflush_line` makes the data cache line coherent by invalidating it. On CRAY T90 systems, `shmem_udcflush_line` makes the data cache line coherent by invalidating the entire data cache. This has the same effect as the `#pragma _CRI suppress` directive (in C) and the `CDIR$ SUPPRESS` directive (in Fortran) when these directives specify a variable name.

NOTES

On CRAY T3D systems, automatic cache invalidation is disabled prior to any user call to `shmem_set_cache_inv` or `shmem_set_cache_line_inv`.

Because hardware or the operating system ensures that the cache never becomes incoherent on CRAY T3E, CRAY Y-MP, CRAY J90, or CRAY C90 systems, all cache-management routines are no-ops on these architectures.

SEE ALSO

`intro_shmem`(3), `shmem_put`(3), `shmem_swap`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_collect, shmem_fcollect, shmem_collect32, shmem_fcollect32, shmem_collect4, shmem_fcollect4 – Concatenates blocks of data from multiple processing elements (PEs) to an array in every PE

SYNOPSIS

C:

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

void shmem_collect(void *target, void *source, int nlong,
   int PE_start, int logPE_stride, int PE_size, long *pSync);

void shmem_fcollect(void *target, void *source, int nlong,
   int PE_start, int logPE_stride, int PE_size, long *pSync);

void shmem_collect32(void *target, void *source, int nlong,
   int PE_start, int logPE_stride, int PE_size, long *pSync);

void shmem_fcollect32(void *target, void *source, int nlong,
   int PE_start, int logPE_stride, int PE_size, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)

CALL SHMEM_COLLECT(target, source, nlong, PE_start, logPE_stride,
   PE_size, pSync)

CALL SHMEM_FCOLLECT(target, source, nlong, PE_start, logPE_stride,
   PE_size, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)

CALL SHMEM_COLLECT4(target, source, nlong, PE_start, logPE_stride,
   PE_size, pSync)

CALL SHMEM_FCOLLECT4(target, source, nlong, PE_start, logPE_stride,
   PE_size, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) collective routines concatenate *nlong* 64- or 32-bit data items from the source array into the target array, over the set of PEs defined by *PE_start*, *log2PE_stride*, and *PE_size*, in processor number order. The resultant target array contains the contribution from PE *PE_start* first, then the contribution from PE *PE_start* + *PE_stride* second, and so on. The collected result is written to the target array for all PEs in the active set.

The resulting target array is as follows:

```
-------------------------------------------------------
   source(1..nlong)
       from PE (PE_start + 0 * (2**logPE_stride))
-------------------------------------------------------
   source(1..nlong)
       from PE (PE_start + 1 * (2**logPE_stride))
-------------------------------------------------------
   ...
-------------------------------------------------------
   source(1..nlong)   from
       PE (PE_start + (PE_size - 1) * (2**logPE_stride))
-------------------------------------------------------
```

The arguments are as follows:

target A symmetric array. The *target* argument must be large enough to accept the concatenation of the source arrays on all PEs. The data types are as follows:

Routine	Data Type
`shmem_collect`, `shmem_fcollect`	Any data type with an element size of 64 bits. Fortran derived types, Fortran character type, and C structures are not permitted.
`shmem_collect32`, `shmem_fcollect32`, `shmem_collect4`, `shmem_fcollect4`	Any data type with an element size of 32 bits. Fortran derived types, Fortran character type, and C structures are not permitted.

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

nlong The number of elements in the source array. *nlong* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

The *nlong* argument must be equal on all PEs for `shmem_fcollect` and `shmem_fcollect32`. The *nlong* argument can be different across PEs for `shmem_collect` and `shmem_collect32`.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` in C or `SHMEM_SYNC_VALUE` in Fortran before any of the PEs in the active set enter `shmem_barrier`().

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

Upon return from a collective routine, the following are true for the local PE:

- The target array is updated.
- The data cache region that is mapped to the target data object is coherent in all receiving PEs.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM collective routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* array can be reused on a subsequent SHMEM collective routine only if none of the PEs in the active set are still processing a prior SHMEM collective routine call that used the same *pSync* array. In general, this may be ensured only by doing some type of synchronization. However, in the special case of SHMEM routines being called with the same active set, you can allocate two *pSync* arrays and alternate between them on successive calls.

Routine `shmem_collect` allows the value of *nlong* to be different on each participating processor. Routine `shmem_fcollect`, however, expects the value of *nlong* to be the same for all processors. Because the value for *nlong* 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.

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

EXAMPLES

Example in C:

```
for (i=0; i < _SHMEM_COLLECT_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier(); /* Wait for all PEs to initialize pSync */
shmem_collect(target, source, 64, pe_start, logPE_stride,
   pe_size, pSync);
```

Example in Fortran:

```
 INTEGER PSYNC(SHMEM_COLLECT_SYNC_SIZE)
 DATA PSYNC /SHMEM_COLLECT_SYNC_SIZE*SHMEM_SYNC_VALUE/

 CALL SHMEM_COLLECT(TARGET, SOURCE, NLONG, ROOT, PE_START,
&  LOGPE_STRIDE, PE_SIZE, PSYNC)
 CALL SHMEM_COLLECT(TARGET, SOURCE, 64, PE_START, LOGPE_STRIDE,
&  PE_SIZE, PSYNC)
```

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_iget, shmem_iput, shmem_iget32, shmem_iput32, shmem_iget4, shmem_iput4 – Transfers strided data between remote and local memory

SYNOPSIS

C:

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

void shmem_iget(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);

void shmem_iput(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);

void shmem_iget32(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);

void shmem_iput32(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);
```

Fortran (CF77 and CF90):

```
INTEGER target_inc, source_inc, len, pe
CALL SHMEM_IGET(target, source, target_inc, source_inc, len, pe);

CALL SHMEM_IPUT(target, source, target_inc, source_inc, len, pe);
```

Fortran 90 (CF90) only:

```
INTEGER target_inc, source_inc, len, pe
CALL SHMEM_IGET4(target, source, target_inc, source_inc, len, pe);

CALL SHMEM_IPUT4(target, source, target_inc, source_inc, len, pe);
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The iget and iput routines transfer strided arrays between local and remote memory.

The shmem_iget, shmem_iget32, and shmem_iget4 routines transfer data from the remote address defined by *pe* and *source* into the local address defined by *target*. The function returns when the data has been loaded into the local data array. On CRAY T3D systems, this function uses remote loads.

The shmem_iput, shmem_iput32, and shmem_iput4 routines transfer data from the local address defined by *source* into the remote address defined by *pe* and *target*. The function returns when the data has been sent from the chip (but it may not yet have arrived at the other processing element (PE)). On CRAY T3D systems, this function uses remote stores.

The arguments are as follows:

target Array to be updated. For `iget` routines, *target* resides on the local PE and need not be remotely accessible. For `iput` routines, *target* resides on the remote PE and must be remotely accessible.

The data types are as follows:

Routine	Data Type
`shmem_iget`, `shmem_iput`	Any data type with an element size of 64 bits. Fortran derived types, Fortran character type, and C structures are not permitted.
`shmem_iget4`, `shmem_iput4`, `shmem_iget32`, `shmem_iput32`	Any data type with an element size of 32 bits. Fortran derived types, Fortran character type, and C structures are not permitted.

source Array to be transferred. This must be a remotely accessible data object. The *source* argument can have any of the data types permitted for *target*.

target_inc The stride between consecutive elements of the *target* array. The stride is scaled by the element size of the *target* array. A value of 1 indicates contiguous data. *target_inc* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

source_inc The stride between consecutive elements of the source array. The stride is scaled by the element size of the *source* array. A value of 1 indicates contiguous data. *source_inc* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

len Number of elements in the *target* and *source* arrays. *len* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pe PE number of the remote PE. *pe* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

EXAMPLES

The `shmem_iget` routine is semantically equivalent to the following:

C example:

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

Fortran example:

```
SRC_IDX = 1
TAR_IDX = 1
DO I = 1, LEN
        CALL SHMEM_GET(TARGET(TAR_IDX), SOURCE(SRC_IDX), 1, PE)
        SRC_IDX = SRC_IDX + SOURCE_INC
        TAR_IDX = TAR_IDX + TARGET_INC
ENDDO
```

NOTES

On CRAY T3D systems, you can use the `put` routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using `shmem_udcflush`(), `shmem_udcflush_line`(), the `#pragma _CRI suppress` directive (from C), or the `CDIR$ SUPPRESS` directive (from Fortran).

See `intro_shmem`(3) and `shmem_put`(3) for more information about remotely accessible data objects.

SEE ALSO

`intro_shmem`(3), `shmem_put`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_ixget, shmem_ixput, shmem_ixget32, shmem_ixput32, shmem_ixget4, shmem_ixput4 – Transfers strided data between remote and local memory

SYNOPSIS

C:

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

void shmem_ixget(void *target, void *source, long *source_index,
  int len, int pe);

void shmem_ixput(void *target, void *source, long *target_index,
  int len, int pe);

void shmem_ixget32(void *target, void *source, short *source_index,
  int len, int pe);

void shmem_ixput32(void *target, void *source, short *target_index,
  int len, int pe);
```

Fortran (CF77 and CF90):

```
INTEGER len, pe
CALL SHMEM_IXGET(target, source, source_index, len, pe);

CALL SHMEM_IXPUT(target, source, target_index(*), len, pe);
```

Fortran 90 (CF90) only:

```
INTEGER len, pe
CALL SHMEM_IXGET4(target, source, source_index, len, pe);

CALL SHMEM_IXPUT4(target, source, target_index, len, pe);
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The ixget and ixput routines gather or scatter data between remote and local memory.

shmem_ixget, shmem_ixget32, and shmem_ixget4 transfer data from the remote indexed array defined by *pe* and *source* into the local array defined by *target*. The function returns when the data has been loaded into the local data array. On CRAY T3D systems, this function uses remote loads.

shmem_ixput, shmem_ixput32, and shmem_ixput4 transfer data from the local address defined by *source* into the remote indexed address defined by *pe* and *target*. The function returns when the data has been sent from the chip (but it may not yet have arrived at the other processing element (PE)). On CRAY T3D systems, this function uses remote stores.

The arguments are as follows:

target Array to be updated. For `ixget` routines, *target* resides on the local PE and need not be remotely accessible. For `ixput` routines, *target* resides on the remote PE and must be remotely accessible.

The data type of *target* varies with the routine being called. When calling from C, see the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_ixget`, `shmem_ixput`	Any noncharacter date type with a storage size of 8 bytes per element.
`shmem_ixget32`, `shmem_ixput32`, `shmem_ixget4`, `shmem_ixput4`	Any noncharacter data type with a storage size of 4 bytes per element.

source A remotely accessible data object that can be of any data type permissible for the *target* argument.

target_index Used on the *ixput* routines. Array of zero-based indexes into the target array. In C, *target_index* is the type implied in the SYNOPSIS section. In Fortran, the type of *target_index* is as follows:

Routine	Data Type
`shmem_ixput`	Integer, with a storage size of 8 bytes per element.
`shmem_ixput4`	Integer, with a storage size of 4 bytes per element.

source_index Used on the `ixget` routines. Array of zero-based indexes into the source array. In C, *target_index* is the type implied in the SYNOPSIS section. In Fortran, the type of *target_index* is as follows:

Routine	Data Type
`shmem_ixget`	Integer, with a storage size of 8 bytes per element.
`shmem_ixget4`	Integer, with a storage size of 4 bytes per element.

len Number of elements to transfer. *len* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pe PE number of the remote PE. *pe* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

NOTES

On CRAY T3D systems, `shmem_ixget` uses the prefetch queue to gather the remote data. Using the prefetch queue gives consistently good performance results independent of the size of the source array or of the distribution of values in the `source_index` array.

On CRAY T3D systems, you can use the `ixput` routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using `shmem_udcflush`(), `shmem_udcflush_line`(), the `#pragma _CRI suppress` directive (from C), or the `CDIR$ SUPPRESS` directive (from Fortran).

See `intro_shmem`(3) and `shmem_put`(3) for more information about remotely accessible data objects.

EXAMPLES

The `shmem_ixget` routine is semantically equivalent to the following:

C:

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

Fortran:

```
DO I = 1, LEN
    CALL SHMEM_GET( TARGET(I), SOURCE(SOURCE_INDEX(I) + 1), 1, PE )
ENDDO
```

SEE ALSO

`intro_shmem`(3), `shmem_put`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_clear_lock, shmem_set_lock, shmem_test_lock – Releases, locks, and tests a mutual exclusion memory lock

SYNOPSIS

C:

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

void shmem_clear_lock(long *lock);

void shmem_set_lock(long *lock);

int shmem_test_lock(long *lock);
```

Fortran (CF77 and CF90):

```
INTEGER lock, SHMEM_TEST_LOCK

CALL SHMEM_CLEAR_LOCK(lock)

CALL SHMEM_SET_LOCK(lock)

I = SHMEM_TEST_LOCK(lock)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shmem_clear_lock routine releases a lock previously set by shmem_set_lock after ensuring that all local and remote stores initiated in the critical region are complete.

The shmem_set_lock routine sets a mutual exclusion lock after waiting for the lock to be freed by any other task currently holding the lock. Waiting tasks are assured of getting the lock in a first-come, first-served manner.

The shmem_test_lock function sets a mutual exclusion lock only if it is currently cleared. By using this function, a task can avoid blocking on a set lock. If the lock is currently set, the function returns without waiting.

These routines are appropriate for protecting a critical region from simultaneous update by multiple tasks. They accept the following argument:

lock A symmetric data object that is a scalar value or an array of length 1. This data object must be set to 0 on all processing elements (PEs) prior to the first use. *lock* must be of type integer. If you are using the CF90 compiler, it must be of default kind.

NOTES

The term *symmetric data object* is defined on intro_shmem(3).

The shmem_clear_lock and shmem_set_lock routines do not accept CRAFT shared variables. Use the CLEAR_LOCK(3F) and SET_LOCK(3F) routines if you want to use CRAFT shared locks. For more information on CRAFT, see the *Cray MPP Fortran Reference Manual*, publication SR–2504.

RETURN VALUES

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

SEE ALSO

`intro_shmem`(3)

`CLEAR_LOCK`(3F), `SET_LOCK`(3F), `TEST_LOCK`(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

NAME

shmem_short_max_to_all, shmem_int_max_to_all, shmem_int8_max_to_all, shmem_comp8_max_to_all, shmem_int4_max_to_all, shmem_real4_max_to_all – Performs a maximum function reduction across a set of processing elements (PEs)

SYNOPSIS

C:

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

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);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

CALL SHMEM_INT8_MAX_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

CALL SHMEM_COMP8_MAX_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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_REAL4_MAX_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see `intro_shmem`(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_max_to_all`	Complex, with an element size equal to two 8-byte real values
`shmem_int8_max_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_max_to_all`	Integer, with an element size of 4 bytes
`shmem_real4_max_to_all`	Real, with an element size of 4 bytes

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the maximum value of real variable `FOO` across all even PEs.

```
      INCLUDE "mpp/shmem.fh"

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

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

Example 2: Consider the following C call:

```
shmem_int_max_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_max_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_max_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_max_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_min_to_all, shmem_int_min_to_all, shmem_float_min_to_all, shmem_double_min_to_all, shmem_int8_min_to_all, shmem_real8_min_to_all, shmem_int4_min_to_all, shmem_real4_min_to_all – Performs a minimum function reduction across a set of processing elements (PEs)

SYNOPSIS

C:

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

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_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);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

CALL SHMEM_INT8_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)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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_REAL4_MIN_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see `intro_shmem`(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_int8_min_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_min_to_all`	Integer, with an element size of 4 bytes
`shmem_real8_min_to_all`	Real, with an element size of 8 bytes
`shmem_real4_min_to_all`	Real, with an element size of 4 bytes

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the minimum value of real variable `FOO` across all the even PEs.

```
 INCLUDE "mpp/shmem.fh"

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

 IF ( MOD(MY_PE(),2) .EQ. 0) THEN
    CALL SHMEM_REAL8_MIN_TO_ALL(FOOMIN, FOO, NR, 0, 1, N$PES/2,
&      PWRK, PSYNC)
    PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOMIN
 ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_min_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_min_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_min_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_min_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_my_pe, shmem_n_pes – Returns processing element (PE) information

SYNOPSIS

C:

```
int  shmem_my_pe(void);

int  shmem_n_pes(void);
```

Fortran (CF77 and CF90):

```
INTEGER  SHMEM_MY_PE,  SHMEM_N_PES

MYPE  =  SHMEM_MY_PE()

NPES  =  SHMEM_N_PES()
```

IMPLEMENTATION

Cray MPP systems

libsma in Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The shmem_my_pe function returns the number of the local PE.

The shmem_n_pes function returns the total number of PEs in the current application.

NOTES

The Cray Research C and CF77 compilers provide the _my_pe() and MY_PE() intrinsic functions on Cray MPP systems, which generate fast inline code to return the current PE. The shmem_my_pe function is most useful on Cray PVP systems and when using the CF90 compiler because no such intrinsic is available.

The Cray Research C and CF77 compilers provide the _num_pes() intrinsic function on Cray MPP systems and N$PES constant, which generate fast inline code to return the number of PEs. The shmem_n_pes function is most useful when using the CF90 compiler because no such intrinsic or constant is available.

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, SR–2196

NAME

shmem_short_or_to_all, shmem_int_or_to_all, shmem_int8_or_to_all, shmem_int4_or_to_all – Performs a logical OR function reduction across a set of processing elements (PEs)

SYNOPSIS

C:

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

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);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

CALL SHMEM_INT8_OR_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_int8_or_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_or_to_all`	Integer, with an element size of 4 bytes

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent in all receiving PEs.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the logical OR of the integer variable `FOO` across all even PEs.

```
INCLUDE "mpp/shmem.fh"

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

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
   CALL SHMEM_INT8_OR_TO_ALL(FOOOR, FOO, NR, 0, 1, N$PES/2,
&      PWRK, PSYNC)
   PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOOR
ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_or_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_or_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_or_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_or_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

`shmem_short_prod_to_all`, `shmem_int_prod_to_all`, `shmem_float_prod_to_all`, `shmem_double_prod_to_all`, `shmem_complexf_prod_to_all`, `shmem_complexd_prod_to_all`, `shmem_int8_prod_to_all`, `shmem_real8_prod_to_all`, `shmem_comp8_prod_to_all`, `shmem_int4_prod_to_all`, `shmem_real4_prod_to_all`, `shmem_comp4_prod_to_all` – Performs a product reduction across a set of processing elements (PEs)

SYNOPSIS

C:

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

void shmem_short_prod_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, short *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_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_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 (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

CALL SHMEM_INT8_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_COMP8_PROD_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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_REAL4_PROD_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

CALL SHMEM_COMP4_PROD_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called and the language used. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_prod_to_all`	Complex, with an element size equal to two 8-byte real values
`shmem_comp4_prod_to_all`	Complex, with an element size equal to two 4-byte real values
`shmem_int8_prod_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_prod_to_all`	Integer, with an element size of 4 bytes
`shmem_real8_prod_to_all`	Real, with an element size of 8 bytes
`shmem_real4_prod_to_all`	Real, with an element size of 4 bytes

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.

- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the product of the real variable `FOO` across all the even PEs.

```
INCLUDE "mpp/shmem.fh"

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

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
   CALL SHMEM_COMP8_PROD_TO_ALL(FOOPROD, FOO, NR, 0, 1, N$PES/2,
&      PWRK, PSYNC)
   PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOPROD
ENDIF
```

Example 2: Consider the following C call:

```
shmem_short_prod_to_all(target, source, 3, 0, 0, 8, pwrk, psync);
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_short_prod_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_short_prod_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_short_prod_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_get, shmem_put, shmem_get32, shmem_put32, shmem_get4, shmem_put4 – Transfers data between local and remote memory

SYNOPSIS

C:

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

void shmem_get(void *target, void *source, int len,
   int pe);

void shmem_put(void *target, void *source, int len,
   int pe);

void shmem_get32(void *target, void *source, int len,
   int pe);

void shmem_put32(void *target, void *source, int len,
   int pe);
```

Fortran (CF77 and CF90):

```
INTEGER len, pe
CALL SHMEM_GET(target, source, len, pe);

CALL SHMEM_PUT(target, source, len, pe);
```

Fortran 90 (CF90) only:

```
INTEGER len, pe
CALL SHMEM_GET4(target, source, len, pe);

CALL SHMEM_PUT4(target, source, len, pe);
```

IMPLEMENTATION

Cray MPP systems

shmem_get and shmem_put are included in the Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The put and get routines provide a high-performance method for transfering data between remote and local memory, but they require careful use of variables and addresses.

shmem_get, shmem_get32, and shmem_get4 transfer data from the remote data object defined by *pe* and *source* into the local data object defined by *target*. The function returns when the data has been stored in the local data object.

shmem_put, shmem_put32, and shmem_put4 transfer data from the local data object defined by *source* into the remote data object defined by *pe* and *target*. The function returns when the data has been sent from the local processing element (PE), but it may not yet have arrived at the other PE.

The arguments are as follows:

target Array to be updated. For `get` routines, *target* resides on the local PE. For `put` routines, *target* resides on the remote PE and must be remotely accessible.

The data type of *target* varies with the version of the `put` routine being called and the language used. The *target* data types are as follows:

Routine	Data Type
`shmem_get`, `shmem_put`	Any noncharacter type that has a storage size equal to 8 bytes.
`shmem_get32`, `shmem_put32`, `shmem_get4`, `shmem_put4`	Any noncharacter type that has a storage size equal to 4 bytes.

source Array to be copied. The *source* argument can have any data type permitted for *target*.

For `put` routines, *source* resides on the local PE. For `get` routines, *source* resides on the remote PE and must be remotely accessible.

len Number of elements in the *target* and *source* arrays. *len* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pe PE number of the remote PE. *pe* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

For the most part, `shmem_get32`() and `shmem_put32`() transfer data at the same rate as their `get` and `put` counterparts. However, performance for the 32-bit routines degrades significantly if one of the target or source arrays is aligned in memory on a 64-bit word boundary and the other is not. You can avoid this problem by cache aligning the source and target arrays and then always beginning transfers with either even-numbered elements of both arrays or odd-numbered elements of both arrays. You should never begin transfers with one even- and one odd-numbered element. The following C example shows this:

```
#pragma _CRI cache_align target, source
short target[256], source[256];

example()
{
        shmem_put32(target, source, nlong, pe);
            /* runs fast   */
        shmem_put32(&target[5], &source[15], nlong, pe);
            /* runs fast   */
        shmem_put32(target, &source[5], nlong, pe);
            /* runs slower */
}
```

EXAMPLES

This simple `shmem_put` example runs on Cray MPP systems and Cray PVP systems.

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

main()
{

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

#if     _CRAY1
```

```
#pragma taskcommon target
#endif
        static long target[10];

#if     _CRAY1
        start_pes(2);
#endif

        if (_my_pe() == 0) {
                /* put 10 words into target on PE 1 */
                shmem_put(target, source, 10, 1);
        }

        barrier();                      /* wait for completion */

        if (_my_pe() == 1) {
                shmem_udcflush();
        }

        printf("target[0] on PE %d is %d0, _my_pe(), target[0]);


}
```

NOTES

On CRAY T3D, you can use the `put` routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using `shmem_udcflush`(), `shmem_udcflush_line`(), the `#pragma _CRI suppress` directive (from C), or the `CDIR$ SUPPRESS` directive (from Fortran).

CRAY J90, CRAY Y-MP, and CRAY C90 systems do not use scalar caches for multitasked programs, so calls to cache management functions are not needed when using the `shmem_put` function.

CRAY T3E systems ensure that data caches stay coherent while doing remote data writes, so calls to cache management functions are not needed when using the `put` routines.

See `intro_shmem`(3) for a definition of the term *remotely accessible*. Note that remote data objects must be declared on a `TASKCOMMON` compiler directive on Cray PVP systems.

On Cray MPP systems, use dynamically allocated data objects with the `put` and `get` routines carefully because:

- Each PE may have a different address for a stack or heap data object.
- These data objects are not remotely accessible in some CRAFT programs.

Stack variables are defined only for the lifetime of the function that declares them, and many variables (located in many functions) can share the same address. For dynamically allocated data, you cannot simply specify a local variable name for the remote address. Instead, you must explicitly transfer the remote address to the PE calling one of these routines. You must also ensure, through your algorithm, that the address remains valid as long as the other PE may be using it. You risk receiving operand range errors if the remote address is not accessible remotely.

These routines perform no synchronization. You must be sure that both PEs are ready for the action to take place. If you use `shmem_get`, for example, you must be sure that the data you are copying from the other PE has been updated or you will pick up old data or partially updated data.

On CRAY T3D systems, `shmem_put` transmits data with about twice the bandwidth of `shmem_get`. Algorithms get higher performance if `shmem_put` is used on the sending processor instead of using `shmem_get` on the receiving processor.

SEE ALSO

`intro_shmem`(3), `shmem_iput`(3), `shmem_ixput`(3), `shmem_quiet`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

`shmem_quiet` – Waits for completion of all outstanding remote writes issued by a processing element (PE)

SYNOPSIS

C:

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

void shmem_quiet(void);
```

Fortran:

```
CALL SHMEM_QUIET
```

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

`shmem_quiet` waits for completion of all outstanding remote writes initiated from the current PE. Remote writes are issued by calls to `shmem_put` and related `put` routines. When control returns from `shmem_put`, the data is delivered to the communication circuitry but has not yet arrived at the remote PE. `shmem_quiet` does not return until all data is delivered to the remote PE's memory.

NOTES

`shmem_quiet` is an inlined macro in C. To get the function form, the `shmem_quiet` macro name must be undefined by use of the `#undef shmem_quiet` `cpp`(1) directive.

`shmem_quiet` is most useful as a way of ensuring ordering of delivery of several `put` operations. For example, you might use `shmem_quiet` to await delivery of a block of data before issuing another `put`, which sets a completion flag in the remote PE.

`shmem_quiet` is not usually needed if `barrier`(3C), `barrier`(3F), `BARSYNC`(3F), or `shmem_barrier`(3) are called. The barrier routines all wait for the completion of outstanding remote writes (`put`s).

EXAMPLES

```
PROGRAM COMPFLAG
INTEGER FLAG_VAR, ARRAY(100), RECEIVER, SENDER
COMMON/FLAG/FLAG_VAR
COMMON/DATA/ARRAY
INTRINSIC MY_PE

FLAG_VAR = 0
CALL BARRIER                    ! wait for FLAG_VAR to be initialized

SENDER = 0                      ! PE 0 sends the data
RECEIVER = 1                    ! PE 1 receives the data
IF (MY_PE() .EQ. 0) THEN
  ARRAY = 33
  CALL SHMEM_PUT(ARRAY, ARRAY, 100, RECEIVER)   ! start sending data
  CALL SHMEM_QUIET                              ! wait for delivery
  CALL SHMEM_PUT(FLAG_VAR, 1, 1, RECEIVER)      ! send completion flag
ELSE IF (MY_PE() .EQ. RECEIVER) THEN
  CALL SHMEM_UDCFLUSH
  CALL SHMEM_WAIT(FLAG_VAR, 0)
  PRINT*,ARRAY                  ! ARRAY has been delivered
ENDIF
END
```

SEE ALSO

intro_shmem(3), shmem_barrier(3), shmem_put(3), shmem_wait(3)

cpp(1)

barrier(3C), barrier(3F), BARSYNC(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_stack – Makes a stack address remotely accessible

SYNOPSIS

C:

```
void shmem_stack(void *stack_var);
```

Fortran (CF77 and CF90):

```
CALL SHMEM_STACK(stack_var)
```

IMPLEMENTATION

CRAY T3D systems

DESCRIPTION

The shmem_stack routine makes a stack address on a remote processing element (PE) remotely accessible from the local PE. This is done by checking that the *stack_var* address is a valid stack address on the current PE. If *stack_var* is beyond the top of stack, shmem_stack extends the stack to include *stack_var*.

The shmem_stack routine can be called prior to shmem_put(3) or shmem_get(3) when the *stack_var* address is on the stack. This ensures that an operand range error does not occur during the data-passing step. For shmem_get(3) or shmem_put(3) to work properly, a remote *stack_var* address must also be a valid address on the local PE.

The shmem_stack routine accepts the following argument:

stack_var Data object of any noncharacter type. It must reside on the private stack.

NOTES

For more information about remotely accessible data objects, see intro_shmem(3).

SEE ALSO

intro_shmem(3), shmem_get(3), shmem_put(3)

NAME

shmem_short_sum_to_all, shmem_int_sum_to_all, shmem_float_sum_to_all, shmem_double_sum_to_all, shmem_complexf_sum_to_all, shmem_complexd_sum_to_all, shmem_int8_sum_to_all, shmem_real8_sum_to_all, shmem_comp8_sum_to_all, shmem_int4_sum_to_all, shmem_real4_sum_to_all, shmem_comp4_sum_to_all – Performs a sum reduction across a set of processing elements (PEs)

SYNOPSIS

C:

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

void shmem_short_sum_to_all(short *target,
   short *source, int nreduce, int PE_start,
   int logPE_stride, int PE_size, short *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_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_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 (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

INTEGER target(nreduce), source(nreduce)
INTEGER pWrk(MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_INT8_SUM_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

REAL target(nreduce), source(nreduce)
REAL pWrk(MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_REAL8_SUM_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

COMPLEX target(nreduce), source(nreduce)
COMPLEX pWrk(MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_COMP8_SUM_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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

INTEGER(KIND=4) target(nreduce), source(nreduce)
INTEGER(KIND=4) pWrk(MAX(nreduce/2+1,
   SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_INT4_SUM_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

REAL(KIND=4) target(nreduce), source(nreduce)
REAL(KIND=4) pWrk(MAX(nreduce/2+1,
   SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_REAL4_SUM_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

COMPLEX(KIND=4) target(nreduce), source(nreduce)
COMPLEX(KIND=4) pWrk(MAX(nreduce/2+1,
   SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_COMP4_SUM_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple processing elements (PEs). A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see `intro_shmem`(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called and the language used. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_sum_to_all`	Complex. If you are using the CF90 compiler, it must be a default complex value.
`shmem_comp4_sum_to_all`	`COMPLEX(KIND=4)`.
`shmem_int8_sum_to_all`	Integer. If you are using the CF90 compiler, it must be a default integer value.
`shmem_int4_sum_to_all`	`INTEGER(KIND=4)`.
`shmem_real8_sum_to_all`	Real. If you are using the CF90 compiler, it must be a default real value.
`shmem_real4_sum_to_all`	`REAL(KIND=4)`.

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the sum of the real variable `FOO` across all even PEs.

```
 INCLUDE "mpp/shmem.fh"

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

 IF ( MOD(MY_PE(),2) .EQ. 0) THEN
    CALL SHMEM_INT4_SUM_TO_ALL(FOOSUM, FOO, NR, 0, 1, N$PES/2,
&      PWRK, PSYNC)
    PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOSUM
 ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_sum_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
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,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_swap – Performs an atomic swap to a remote address

SYNOPSIS

C:

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

long shmem_swap(long *target, long swaperand, int pe);
```

Fortran:

```
INTEGER SHMEM_SWAP
INTEGER target, swaperand, pe, ires

ires = SHMEM_SWAP(target, swaperand, pe)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

shmem_swap performs an atomic swap operation. It writes the 64-bit value *swaperand* into *target* on processing element (PE) *pe* and returns the previous contents of *target* as an atomic operation.

The arguments are as follows:

target The remotely accessible integer data object to be updated on the remote PE. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *target* should match that implied in the SYNOPSIS section.

swaperand The integer value to be swapped into the *target* address on the remote PE. If you are using the CF90 compiler, it must be a default integer value.

pe An integer that indicates the PE number on which *target* is to be updated. If you are using the CF90 compiler, it must be a default integer value.

NOTES

The term *remotely accessible* is defined in intro_shmem(3).

shmem_swap writes directly into another PE's memory without notifying that PE, so the receiving PE should normally set automatic cache invalidation by use of shmem_set_cache_inv(3) or shmem_set_cache_line_inv(3). If this is not done, the receiving PE's data cache might become incoherent.

You can use shmem_swap without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using shmem_udcflush(), shmem_udcflush_line(), the #pragma _CRI suppress directive (from C), or the CDIR$ SUPPRESS directive (from Fortran).

RETURN VALUES

The contents that had been at the target address on the remote PE prior to the swap.

SEE ALSO

`intro_shmem`(3), `shmem_clear_cache_inv`(3), `shmem_set_cache_inv`(3), `shmem_set_cache_line_inv`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_wait, shmem_wait_until – Waits for a variable on the local processing element (PE) to change

SYNOPSIS

C:

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

void shmem_wait( long *ivar, long cmp_value )

void shmem_wait_until( long *ivar, int cmp, long value )
```

Fortran:

```
include ´mpp/shmem.fh´

INTEGER ivar, cmp, cmp_value

CALL SHMEM_WAIT(ivar, cmp_value)

CALL SHMEM_WAIT_UNTIL(ivar, cmp, cmp_value)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

shmem_wait and shmem_wait_until wait for *ivar* to be changed by a remote write or atomic swap issued by a different processor. These routines can be used for point-to-point directed synchronization. A call to shmem_wait does not return until some other processor writes a value, not equal to *cmp_value*, into *ivar* on the waiting processor. A call to shmem_wait_until does not return until some other processor changes *ivar* to satisfy the condition implied by *cmp* and *cmp_value*. This mechanism is useful when a processor needs to tell another processor that it has completed some action.

The arguments are as follows:

ivar A remotely accessible integer variable that is being updated by a remote processor. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *ivar* should match that implied in the SYNOPSIS section.

cmp The compare operator that compares *ivar* with *cmp_value*. *cmp* must be of type integer. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *cmp* should match that implied in the SYNOPSIS section.

The following *cmp* values are supported:

cmp **Value**	**Comparison**
SHMEM_CMP_EQ (Fortran), _SHMEM_CMP_EQ (C)	Equal
SHMEM_CMP_NE (Fortran), _SHMEM_CMP_NE (C)	Not equal
SHMEM_CMP_GT (Fortran), _SHMEM_CMP_GT (C)	Greater than
SHMEM_CMP_LE (Fortran), _SHMEM_CMP_LE (C)	Less than or equal to
SHMEM_CMP_LT (Fortran), _SHMEM_CMP_LT (C)	Less than

`SHMEM_CMP_GE` (Fortran), `_SHMEM_CMP_GE` (C) Greater than or equal to

cmp_value *cmp_value* must be of type integer. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *cmp_value* should match that implied in the SYNOPSIS section.

The `shmem_wait` routine returns when *ivar* is no longer equal to *cmp_value*.

The `shmem_wait_until` routine returns when the compare condition is true. The compare condition is defined by the *ivar* argument compared with the *cmp_value* using the comparison operator, *cmp*.

NOTES

`shmem_wait` and `shmem_wait_until` perform the following functions:

- Invalidate the cache line associated with *ivar*
- Update the cache invalidation filter so that remote writes to this location will invalidate the corresponding cache line
- Wait, by looping in cache, until the value of *ivar* satisfies the wait condition

It may be more efficient to write a simple spin-wait loop with a variable that is declared or cast volatile. This is the case if other processors will not be writing blocks of data into the waiting processor. On CRAY T3D systems, it takes `SHMEM_WAIT`() about 100 clock periods (CPs) to set up the cache invalidation filter and another 100 CPs to disable the filter before returning to the caller.

EXAMPLES

Example 1. The following call returns when variable *ivar* is not equal to 100:

```
CALL SHMEM_WAIT(ivar, 100)
```

Example 2. The following call to `SHMEM_WAIT_UNTIL` is equivalent to the call to `SHMEM_WAIT` in example 1:

```
CALL SHMEM_WAIT_UNTIL(ivar, SHMEM_CMP_NE, 100)
```

Example 3. The following C call waits until the sign bit in `ivar` is set by a transfer from a remote PE:

```
shmem_wait_until(&ivar, _SHMEM_CMP_LT, 0);
```

Example 4. The following Fortran example is in the context of a subroutine:

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

SEE ALSO

`intro_shmem`(3), `shmem_put`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_xor_to_all, shmem_int_xor_to_all, shmem_int8_xor_to_all, shmem_real8_xor_to_all, shmem_comp8_xor_to_all, shmem_int4_xor_to_all, shmem_real4_xor_to_all, shmem_comp4_xor_to_all – Performs a reduction across a set of processing elements (PEs)

SYNOPSIS

C:

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

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);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

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

CALL SHMEM_INT8_XOR_TO_ALL(target, source,
   nreduce, PE_start, logPE_stride, PE_size,
   pWrk, pSync)

CALL SHMEM_REAL8_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)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

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_REAL4_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)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

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

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

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called and the language used. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
shmem_comp8_xor_to_all	Complex, with an element size equal to two 8-byte real values
shmem_comp4_xor_to_all	Complex, with an element size equal to two 4-byte real values
shmem_int8_xor_to_all	Integer, with an element size of 8 bytes
shmem_int4_xor_to_all	Integer, with an element size of 4 bytes
shmem_real8_xor_to_all	Real, with an element size of 8 bytes
shmem_real4_xor_to_all	Real, with an element size of 4 bytes

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

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In 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.

pSync A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

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

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

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

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction 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 a SHMEM 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.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and computes the exclusive OR of variable FOO across all even PEs.

```
INCLUDE "mpp/shmem.fh"

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

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

Example 2: Consider the following C call:

```
shmem_short_xor_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_short_xor_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_short_xor_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_short_xor_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

`start_pes` – Starts a processing element (PE) for a message- or data-passing program

SYNOPSIS

C:

`void start_pes(int` *npes*`)`

Fortran:

`CALL START_PES(`*npes*`)`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `start_pes` routine starts up additional tasks that enter at a main program. This routine starts multitasked tasks, also called PEs, for use in standalone shared memory (SHMEM), Parallel Virtual Machine (PVM), or Message Passing Interface (MPI) applications.

The `start_pes` routine can be called only once by the master task, or PE 0. It is a no-op if any other PE calls this function. This makes it convenient to simply place a call to `start_pes` at the first line of the main program to set up a program to be a message-passing or data-passing application.

The `start_pes` routine accepts the following argument:

npes Identifies the total number of PEs desired. If *npes* is 0, then the number of PEs is selected by the `NPES` environment variable.

EXAMPLES

This is a simple two-PE program that calls `shmem_put`(3):

```
PROGRAM PUT

INTEGER TARG, SRC, RECEIVER, BAR
TASK COMMON /T/ TARG
PARAMETER (RECEIVER=1)
PARAMETER (NPES=2)
INTEGER SHMEM_MY_PE

CALL START_PES(NPES)

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

CALL BARRIER       ! synchronize sender and receiver

IF (SHMEM_MY_PE() .EQ. RECEIVER) THEN
        CALL SHMEM_UDCFLUSH()
        PRINT*,'PE ',SHMEM_MY_PE(),' TARG=',TARG
        IF (TARG.NE.33) PRINT*,'FAIL'
ENDIF
END
```

NOTES

The `start_pes` routine uses a barrier to synchronize the newly started tasks before any of the new tasks enter the main program.

SEE ALSO

`intro_shmem`(3), `num_pes`(3), `my_pe`(3I), `shmem_put`(3)

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196