QUADRICS

Shmem Programming Manual

Quadrics Supercomputers World Ltd. **Document Version 3 - June 27th 2001**

The Quadrics Supercomputers World Ltd. (Quadrics) web site can be found at:

```
http://www.quadrics.com/
```

Quadrics' address is:

QSW Limited
One Bridewell Street
Bristol
BS1 2AA
UK

Tel: +44-(0)117-9075375
Fax: +44-(0)117-9075395

Circulation Control: None

Document Revision History

Revision	Date	Author	Remarks
1	Dec 2000	BB	Initial revision
2	Jan 2001	DR	First public draft
3	June 2001	RMC	Corrections for Linux release

Contents

List of Tables

1

Preface

1.1 Scope of Manual

This manual describes the Shmem programming library. This library supports a shared-memory programming model where cooperating processes exchange data by performing read and write operations on logically shared variables.

1.2 Audience

This manual is intended for developers who want to develop parallel applications using a shared-memory programming model.

The manual assumes that the reader is familiar with the following:

- UNIX operating system
- C programming language

1.3 Using this Manual

This manual contains three chapters. Their contents are as follows:

Chapter 1 (*Preface*)

describes the layout of the manual and the conventions used to present information

Chapter 2 (*The Shmem Library*)

describes the functions in the Shmem library

Chapter 3 (*Programming Examples*)

contains a worked example of using the Shmem libraries

1.4 Related Information

The following manuals provide additional information relevant to developing parallel applications using Shmem:

- *Elan Programming Manual*
- *RMS Reference Manual*
- *RMS User Manual*

Programming examples are installed in the directory `/usr/lib/rms/examples` (or `/opt/rms/examples` for Solaris) together with makefiles for compiling the programs.

1.5 Location of Online Documentation

Online documentation in HTML format is installed in the directory `/usr/lib/rms/docs/html` (or `/opt/rms/docs/html` for Solaris) and can be accessed from a browser at `http://rmshost:8081/html/index.html`. PostScript and PDF versions of the documents are in `/usr/lib/rms/docs` (or `/opt/rms/docs` for Solaris). Please consult your system administrator if you have difficulty accessing the documentation.

New versions of this and other Quadrics documentation can be found on the Quadrics web site `http://www.quadrics.com`.

1.6 Reader's Comments

If you would like to make any comments on this or any other Quadrics manual, please send them to `support@quadrics.com`.

1.7 Conventions

The following typographical conventions have been used in this document:

`monospace type`

Monospace type denotes literal text. This is used for command descriptions, file names and examples of output.

`bold monospace type`

Bold monospace type indicates text that the user enters when contrasted with on-screen computer output.

`italic monospace type`

Italic (slanted) monospace type denotes some meta text. This is used most often in command or parameter descriptions to show where a textual value is to be substituted.

italic type	Italic (slanted) proportional type is used in the text to introduce new terms. It is also used when referring to labels on graphical elements such as buttons.
`Ctrl/x`	This symbol indicates that you hold down the `Ctrl` key while you press another key or mouse button (shown here by `x`).
TLA	Small capital letters indicate an abbreviation (see Glossary).
`ls(1)`	A cross-reference to a reference page includes the appropriate section number in parentheses.
#	A number sign represents the superuser prompt.
%, $	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.

2

The Shmem Library

2.1 Introduction

This chapter describes in detail the functions belonging to the Shmem programming library. This library allows user to write parallel applications using a shared-memory programming model, where all the processes can operate on a globally accessible address space. In order to support this programming model, the Shmem routines supply remote data transfer, work-shared broadcast and reduction, barrier synchronization, and atomic memory operations. Furthermore the Shmem routines minimize the overhead associated with data passing requests, maximize bandwidth, and minimize data latency. The Shmem library can be used in conjunction with or as a replacement for message passing routines (e.g. MPI), so that developers can optimally mix message-passing and shared-memory programming models in the same application.

2.2 Compiling

To use the functions in the Shmem library, programs must include the header file `shmem.h`. The library functions reference header files which are, by default, installed in the directory `/usr/include` (or `/opt/rms/include` for Solaris.

Programs must be linked with `libshmem.so`. An example command line to compile a program `prog.c` is shown here.

```
cc -o prog prog.c -lshmem
```

Definitions for the Fortran interface to Shmem can be found in the header file `shmem.fh`.

2.3 Using the Shmem Library

Shmem routines can be used in programs that perform computations in separate address spaces and that explicitly pass data to and from different processes in the program. The processes participating in shared memory applications will be referred as *processing elements* (PEs). Typically, target or source data that reside on remote processing elements are identified by passing the address of the corresponding data object on the local PE. The local existence of a corresponding data object implies that a data object is *remotely accessible*. The remotely accessible data object are listed below:

1. Non-stack C and C++ variables.
2. C and C++ data allocated by `malloc()`.
3. C and C++ data allocated by `elan_allocMain()` or `elan_gallocMain()`.

Warning

Note that calls to `malloc()`, `calloc()`, etc are unsynchronised and that these functions are called from other C library routines. You should not rely on dynamically allocated objects being at the same address in each process.

The global allocator `elan_gallocMain()` performs synchronised storage allocation, see *Elan Programming Manual* for details.

2.3.1 Word Lengths

The Shmem library provides functions that perform the same operation for different data types, for example, `shmem_int_put`, `shmem_long_put` and `shmem_double_put`. Some types have different lengths under different operating systems and compiler combinations and in particular they may differ from the lengths found in the Cray Shmem implementation.

The sizes of each type (in bytes) are listed in Table 2.1:

Table 2.1: Data Type Sizes

Type	Tru64 UNIX	Alpha Linux	Solaris	Unicos
`int`	4	4	4	8
`long`	8	8	4	8
`longlong`	8	8	8	16
`float`	4	4	4	8
`double`	8	8	4	8
`longdouble`	16	8	8	16

2.4 Library Function Categories

The functions in the Shmem library, can be grouped according to the operations they perform. These groups are:

Initialisation The initialisation functions (Section 2.5) prepare for the process to participate in shared memory operations. Furthermore this group of functions can be used to retrieve information such as the number of processes elements (PEs) belonging to a shared memory application and the PE identifier.

Remote Write Operations

The Shmem library offers a wide number of functions to perform remote write operations (*put operation*) (Section 2.6) Using these functions a processing element is able to transfer a remotely accessible data object to a remote PE.

Remote Read Operations

The Shmem library offers a wide number of functions to perform remote read operations (*get operation*) (Section 2.7). Using these functions a processing element is able to transfer a remotely accessible data object from a remote PE.

Synchronisation Operations

The library supplies a set of functions providing synchronisation (Section 2.8) among the processing elements participating to a parallel computation. In particular there are two type of synchronisation supported: one is used to express a barrier of groups of PE and the other one is used to notify a PE when a local variable has been modified by a remote PE.

Atomic Memory Operations

The Shmem library supplies programmers with a set of functions allowing *atomic operation* on shared variables (Section 2.9). An atomic memory operation is an atomic (i.e. that cannot be interrupted) read-and-update operation on a remote data object. The value read is guaranteed to be the value of the data object just prior to the update. A wide range of atomic operations are supported like `swap`, `add`, `fetch-and-increment` and `fetch-and-add`

Collective Reduction

The shared memory reduction routines distribute work across a set of PEs (Section 2.10). In particular these functions perform an associative binary operation across a set of values distributed on a set of PEs.

Collective Communication

The shared memory collective routines operate on the same data object on multiple PEs. The Shmem library supplies routines to broadcast a block of data from a processing element to one or more target PEs and to concatenate data item coming from a subset of PEs (Section 2.11).

Address Manipulation

The Shmem library routines that provide multi-process programs with access a contiguous region of virtual address space (Section 2.12) are not supported in this implementation.

Control Data Cache

These routines are supplied for compatibility with the Cray Shmem library and they are implemented as NOPs (Section 2.13).

The following sections describe these groups of functions in more detail. Each section starts by discussing how the functions work as a group and then the functions are described individually.

2.5 Initialisation

The initialisation functions are listed in Table 2.2.

Table 2.2: Initialisation Functions

Name	Description
`start_pes`	Not supported in this implementation.
`shmem_init`	Initialize a process to use the Shmem
`num_pes`	Return the number of processes using Shmem
`my_pe`	Return the processing element identifier

These functions are used to initialize the environment for the processes using the features offered by the Shmem library. In particular the `shmem_init` expects all of the processes to have been started by RMS. The function initialises the caller and then synchronises the caller with the other processes. The functions `num_pes` and `my_pe` supply the number of PEs belonging to the parallel application and the PE identifier of the calling process respectively. The initialisation functions are described in detail on the following pages.

The function `start_pes` is not supported in this implementation. Shmem programs are started via `prun`, see *RMS User Manual* for details.

NAME

my_pe – returns the processing element number of the calling PE

SYNOPSIS

```
#include <shmem.h>
int my_pe(void);
```

DESCRIPTION

The function `my_pe` returns the processing element (PE) number of the calling PE.

RETURN VALUES

The function `my_pe` returns an integer between `0` and `npes-1` where `npes` is the total number of PE's executing the current program.

SEE ALSO

num_pes(3), **shmem_init**(3)

NAME

num_pes – returns the number of PEs running in an application

SYNOPSIS

```
#include <shmem.h>
int num_pes(void);
```

DESCRIPTION

The function num_pes computes the number of PEs running in a parallel application.

RETURN VALUES

The function my_pe returns an integer indicating the number of PEs that are currently allowed to cooperate using the Shmem library functions.

SEE ALSO

my_pe(3), **shmem_init**(3)

NAME

shmem_init – initialise a process to use the Shmem library

SYNOPSIS

```
#include <shmem.h>
void shmem_init(void);
```

DESCRIPTION

The function `shmem_init` initialises the Shmem library. The `shmem_init` call must me made before any other Shmem library calls. The function `shmem_init` should only be called once for each process.

SEE ALSO

num_pes(3), **my_pe**(3)

2.6 Remote Write Operations

The remote write functions are listed in Table 2.3.

Table 2.3: Remote Write Functions

Name	Description
shmem_double_p	Transfers a double data item to a PE
shmem_float_p	Transfers a float data item to a PE
shmem_int_p	Transfers a integer data item to a remote PE
shmem_long_p	Transfers a long data item to a PE
shmem_short_p	Transfers a short data item to a PE
shmem_double_put	Transfers contiguous double data to a PE
shmem_float_put	Transfers contiguous float data to a PE
shmem_int_put	Transfers contiguous integer data to a PE
shmem_long_put	Transfers contiguous long data to a remote PE
shmem_longdouble_put	Transfers contiguous long double data to a PE
shmem_longlong_put	Transfers contiguous long long data to a PE
shmem_short_put	Transfers contiguous short data to a PE
shmem_put	Transfer data type having 64 bits storage size
shmem_put32	Transfers data type having 32 bits storage size
shmem_put64	Transfers data type having 64 bits storage size
shmem_put128	Transfers data type having 128 bits storage size
shmem_putmem	Transfer any contiguous data type to a remote PE
shmem_double_iput	Transfer strided array of double to a remote PE
shmem_float_iput	Transfer strided array of float to a remote PE
shmem_int_iput	Transfer strided array of integer to a remote PE
shmem_long_iput	Transfer strided array of long to a remote PE
shmem_longdouble_iput	Transfer strided array of long double to a PE
shmem_longlong_iput	Transfer strided array of long long to a PE
shmem_short_iput	Transfer strided array of short to a remote PE
shmem_iput	Transfer strided data having 64 bits storage size
shmem_iput32	Transfer strided data having 32 bits storage size
shmem_iput64	Transfer strided data having 64 bits storage size
shmem_iput128	Transfer strided data having 128 bits storage size

These functions provide low latency writes to variables in the memory of a remote PE. The library offers a wide number of remote write functions that are optimized for most of basic data type. In particular the remote write function can be grouped as follows:

1. Functions transferring a single data item having basic type in to the memory of a remote PE (e.g. shmem_double_p, etc.).
2. Functions transferring contiguous data in to the memory of a remote PE (e.g. shmem_double_put, etc.).
3. Functions transferring strided data in to the memory of a remote PE (e.g. shmem_double_iput, etc.).

The remote write functions are described in detail on the following pages.

NAME

shmem_double_p, shmem_float_p, shmem_int_p, shmem_long_p, shmem_short_p – transfer one data item to a remote PE

SYNOPSIS

```
#include <shmem.h>
void shmem_double_p(double *addr, double value, int pe);
void shmem_float_p(float *addr, float value, int pe);
void shmem_int_p(int *addr, int value, int pe);
void shmem_long_p(long *addr, long value, int pe);
void shmem_short_p(short *addr, short value, int pe);
```

PARAMETERS

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

DESCRIPTION

These routines provide a very low latency remote write capability for single elements of most basic types. These functions start the remote transfer and may return before the data is delivered to the remote PE. Use `shmem_quiet()` to force completion on all remote transfers.

The function `shmem_double_p()` transfers a `double` data item to the remote PE.

The function `shmem_float_p()` transfers a `float` data item to the remote PE.

The function `shmem_int_p()` transfers an `integer` data item to the remote PE.

The function `shmem_long_p()` transfers a `long` data item to the remote PE.

The function `shmem_short_p()` transfers a `short` data item to the remote PE.

SEE ALSO

shmem_put(3), **shmem_quiet**(3)

NAME

shmem_put, shmem_double_put, shmem_float_put, shmem_int_put, shmem_long_put, shmem_longdouble_put, shmem_longlong_put, shmem_short_put, shmem_put32, shmem_put64, shmem_put128, shmem_putmem – transfer data to a remote PE

SYNOPSIS

```
#include <shmem.h>
void shmem_put(void *target, const void *source, size_t len,
               int pe);
void shmem_double_put(double *target, const double *source,
                      size_t len, int pe);
void shmem_float_put(float *target, const float *source,
                     size_t len, int pe);
void shmem_int_put(int *target, const int *source, size_t len,
                   int pe);
void shmem_long_put(long *target, const long *source, size_t len,
                    int pe);
void shmem_longdouble_put(long double *target,
                          const long double *source, size_t len,
                          int pe);
void shmem_longlong_put(long long *target,
                        const long long *source, size_t len,
                        int pe);
void shmem_put32(void *target, const void *source, size_t len,
                 int pe);
void shmem_put64(void *target, const void *source, size_t len,
                 int pe);
void shmem_put128(void *target, const void *source, size_t len,
                  int pe);
void shmem_putmem(void *target, const void *source, size_t len,
                  int pe);
void shmem_short_put(short *target, const short *source,
                     size_t len, int pe);
```

PARAMETERS

`target`	The remotely accessible array data object to be updated on the remote PE.
`source`	Data object containing the data to be copied on the remote PE.
`len`	Number of elements in the `target` and `source`. `len` must be of

integer type.

`pe` The number of the remote PE where the data object `source` will be transferred.

DESCRIPTION

These routines provide the means for copying a contiguous data object from the local PE to a contiguous data object on another PE. The routines return when the data has been copied out of the `source` array on the local PE, but not necessarily before the data has been delivered to the remote data object. Use `shmem_quiet()` to force completion on all remote transfers.

The function `shmem_put()` writes any non character type that has a storage size equal to 64 bits to the remote PE.

The function `shmem_double_put()` writes contiguous elements of `double` type to the remote PE.

The function `shmem_float_put()` writes contiguous elements of `float` type to the remote PE.

The function `shmem_int_put()` writes contiguous elements of type `integer` to the remote PE.

The function `shmem_long_put()` write contiguous elements of `long` type to the remote PE.

The function `shmem_longdouble_put()` writes contiguous elements of `long doubletype` to the remote PE.

The function `shmem_longlong_put()` writes contiguous elements of `long long` type to the remote PE.

The function `shmem_short_put()` writes contiguous elements of `short` type to the remote PE.

The function `shmem_put32()` writes any non character type that has a storage size equal to 32 bits to the remote PE.

The function `shmem_put64()` writes any non character type that has a storage size equal to 64 bits to the remote PE.

The function `shmem_put128()` writes any non character type that has a storage size equal to 128 bits to the remote PE.

The function `shmem_putmem()` writes any data type to the remote PE. `len` is scaled in bytes.

SEE ALSO

`shmem_iput(3)`, **`shmem_quiet(3)`**

NAME

shmem_iput, shmem_double_iput, shmem_float_iput, shmem_int_iput, shmem_iput32, shmem_iput64, shmem_iput128, shmem_long_iput, shmem_longdouble_iput, shmem_longlong_iput, shmem_short_iput – transfer strided data to a remote PE

SYNOPSIS

```
#include <shmem.h>
void shmem_iput(void *target, const void *source, ptrdiff_t tst,
                ptrdiff_t sst, size_t len, int pe);
void shmem_double_iput(double *target, const double *source,
                       ptrdiff_t tst, ptrdiff_t sst, size_t len,
                       int pe);
void shmem_float_iput(float *target, const float *source,
                      ptrdiff_t tst, ptrdiff_t sst, size_t len,
                      int pe);
void shmem_int_iput(int *target, const int *source, ptrdiff_t tst,
                    ptrdiff_t sst, size_t len, int pe);
void shmem_iput32(void *target, const void *source, ptrdiff_t tst,
                  ptrdiff_t sst, size_t len, int pe);
void shmem_iput64(void *target, const void *source, ptrdiff_t tst,
                  ptrdiff_t sst, size_t len, int pe);
void shmem_iput128(void *target, const void *source,
                   ptrdiff_t tst, ptrdiff_t sst, size_t len,
                   int pe);
void shmem_long_iput(long *target, const long *source,
                     ptrdiff_t tst, ptrdiff_t sst, size_t len,
                     int pe);
void shmem_longdouble_iput(long double *target,
                           const long double *source,
                           ptrdiff_t tst, ptrdiff_t sst,
                           size_t len, int pe);
void shmem_longlong_iput(long long *target,
                         const long long *source, ptrdiff_t tst,
                         ptrdiff_t sst, size_t len, int pe);
void shmem_short_iput(short *target, const short *source,
                      ptrdiff_t tst, ptrdiff_t sst, size_t len,
                      int pe);
```

PARAMETERS

`target`	The remotely accessible array data object to be updated on the remote PE.
`source`	Array containing the data to be copied on the remote PE.
`tst`	The stride between consecutive elements of the `target` array. The stride is scaled by the element size of the `target` array. A value of `1` indicates contiguous data. `tst` must be of type integer.
`sst`	The stride between consecutive elements of the `source` array. The stride is scaled by the element size of the `source` array. A value of `1` indicates contiguous data. `sst` must be of type integer.
`len`	Number of elements in the `target` and `source`. `len` must be of integer type.
`pe`	The number of the remote PE were strided data will be stored.

DESCRIPTION

These routines provide the means for copying a strided array from the local PE to a contiguous data object on a different PE. The routines return when the data has been copied out of the `source` array on the local PE, but not necessarily before the data has been delivered to the remote data object.

The function `shmem_iput()` writes strided array where each element is any non character type that has a storage size equal to 64 bits to the remote PE.

The function `shmem_double_iput()` writes strided array of type `double` to the remote PE.

The function `shmem_float_iput()` writes strided array of type `float` to the remote PE.

The function `shmem_int_iput()` writes strided array of type `integer` to the remote PE.

The function `shmem_iput32()` writes any non character type that has a storage size equal to 32 bits to the remote PE.

The function `shmem_iput64()` writes strided array where each element is any non character type that has a storage size equal to 64 bits to the remote PE.

The function `shmem_iput128()` writes strided array where each element is any non character type that has a storage size equal to 128 bits to the remote PE.

The function `shmem_long_iput()` writes strided array of type `long` to the remote PE.

The function `shmem_longdouble_iput()` writes strided array of type `long double` to the remote PE.

The function `shmem_longlong_iput()` writes strided array of type `long long` to the remote PE.

The function `shmem_short_iput()` writes strided array of type `short` to the remote PE.

SEE ALSO

shmem_put(3), **shmem_get**(3), **shmem_iget**(3), **shmem_quiet**(3)

2.7 Remote Read Operations

The Shmem library includes the functions shown in Table 2.4 for performing remote read operations:

Table 2.4: Remote Read Functions

Name	Description
shmem_double_g	Transfers a double data item from a PE
shmem_float_g	Transfers a float data item from a PE
shmem_int_g	Transfers a integer data item from a PE
shmem_long_g	Transfers a long data item from a PE
shmem_short_g	Transfers a short data item from a PE
shmem_double_get	Transfers contiguous double data from a PE
shmem_float_get	Transfers contiguous float data from a PE
shmem_int_get	Transfers contiguous integer data from a PE
shmem_long_get	Transfers contiguous long data from a PE
shmem_longdouble_get	Transfers contiguous long double data from a PE
shmem_longlong_get	Transfers contiguous long long data from a PE
shmem_short_get	Transfers contiguous short data from a PE
shmem_get	Transfers data type having 64 bits storage size
shmem_get32	Transfers data type having 32 bits storage size
shmem_get64	Transfers data type having 64 bits storage size
shmem_get128	Transfers data type having 128 bits storage size
shmem_getmem	Transfers any contiguous data type from a remote PE
shmem_double_iget	Transfer strided array of double from a remote PE
shmem_float_iget	Transfer strided array of float from a remote PE
shmem_int_iget	Transfer strided array of integer from a remote PE
shmem_long_iget	Transfer strided array of long from a remote PE
shmem_longdouble_iget	Transfer strided array of long double from a remote PE
shmem_longlong_iget	Transfer strided array of long long from a remote PE
shmem_short_iget	Transfer strided array of short from a remote PE
shmem_iget	Transfer strided data having 64 bits storage size
shmem_iget32	Transfer strided data having 32 bits storage size
shmem_iget64	Transfer strided data having 64 bits storage size
shmem_iget128	Transfer strided data having 128 bits storage size

These functions provide low latency reads of variables stored in the memory of a remote PE. The library offers a wide number of remote read (*get*) functions that are optimized for most basic data types. In particular the remote read functions can be grouped as follows:

1. Functions reading a single data item having basic type from the memory of a remote PE (e.g. shmem_double_g, etc.).
2. Functions reading contiguous data from from the memory of a remote PE (e.g. shmem_double_get, etc.).

3. Functions reading strided data from the memory of a remote PE (e.g. `shmem_double_iget`, etc.).

The remote read functions are described in detail on the following pages.

NAME

shmem_double_g, shmem_float_g, shmem_int_g, shmem_long_g, shmem_short_g – transfer one data item from a remote PE

SYNOPSIS

```
#include <shmem.h>

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

PARAMETERS

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

DESCRIPTION

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

The function `shmem_double_g()` transfers a `double` data item from a remote PE.

The function `shmem_float_g()` transfers a `float` data item from a remote PE.

The function `shmem_int_g()` transfers a `integer` data item from a remote PE.

The function `shmem_long_g()` transfers a `long` data item from a remote PE.

The function `shmem_short_g()` transfers a `short` data item from a remote PE.

RETURN VALUES

These functions return the contents that had been at the target address `addr` on the remote PE specified by `pe`.

SEE ALSO

shmem_get(3)

NAME

shmem_get, shmem_double_get, shmem_float_get, shmem_get32, shmem_get64, shmem_get128, shmem_getmem, shmem_int_get, shmem_long_get, shmem_longdouble_get, shmem_longlong_get, shmem_short_get – transfer contiguous data from a remote PE

SYNOPSIS

```
#include <shmem.h>
void shmem_get(void *target, const void *source, size_t len,
               int pe);
void shmem_double_get(double *target, const double *source,
                      size_t len, int pe);
void shmem_float_get(float *target, const float *source,
                     size_t len, int pe);
void shmem_get32(void *target, const void *source, size_t len,
                 int pe);
void shmem_get64(void *target, const void *source, size_t len,
                 int pe);
void shmem_get128(void *target, const void *source, size_t len,
                  int pe);
void shmem_getmem(void *target, const void *source, size_t len,
                  int pe);
void shmem_int_get(int *target, const int *source, size_t len,
                   int pe);
void shmem_long_get(long *target, const long *source, size_t len,
                    int pe);
void shmem_longdouble_get(long double *target,
                          const long double *source, size_t len,
                          int pe);
void shmem_longlong_get(long long *target,
                        const long long *source, size_t len,
                        int pe);
void shmem_short_get(short *target, const short *source,
                     size_t len, int pe);
```

PARAMETERS

`target`	Local data object to be updated.
`source`	Data object on the PE identified by `pe` that contains the data to be copied. This data object must be remotely accessible
`len`	Number of elements in the `target` and `source`.

`pe`	The number of the remote PE on which `source` resides.

DESCRIPTION

These routines provide the means for copying a contiguous data object from a remote PE to a contiguous data object in to the local PE. The routines return when the data has been delivered to the `target` array on the local PE.

The function `shmem_get()` reads any non-character type that has a storage size equal to 64 bits from a remote PE.

The function `shmem_double_get()` reads contiguous elements of type `double` from a remote PE.

The function `shmem_float_get()` reads contiguous elements of type `float` from a remote PE.

The function `shmem_get32()` reads any non-character type that has a storage size equal to 32 bits from a remote PE.

The function `shmem_get64()` reads any non-character type that has a storage size equal to 64 bits from a remote PE.

The function `shmem_get128()` reads any non-character type that has a storage size equal to 128 bits from a remote PE.

The function `shmem_getmem()` reads any data type from a remote PE. `len` is scaled in bytes.

The function `shmem_int_get()` reads contiguous elements of type `integer` from a remote PE.

The function `shmem_long_get()` reads contiguous elements of type `long` from a remote PE.

The function `shmem_longdouble_get()` reads contiguous elements of type `long double` from a remote PE.

The function `shmem_longlong_get()` reads contiguous elements of type `long long` from a remote PE.

The function `shmem_short_get()` reads contiguous elements of type `short` from a remote PE.

SEE ALSO

`shmem_iput(3)`, **`shmem_put(3)`**, **`shmem_iget(3)`**, **`shmem_quiet(3)`**

NAME

shmem_iget, shmem_double_iget, shmem_float_iget, shmem_iget32, shmem_iget64, shmem_iget128, shmem_int_iget, shmem_long_iget, shmem_longdouble_iget, shmem_longlong_iget, shmem_short_iget – transfer strided data from a remote PE

SYNOPSIS

```
#include <shmem.h>
void shmem_iget(void *target, const void *source, ptrdiff_t tst,
                ptrdiff_t sst, size_t len, int pe);
void shmem_double_iget(double *target, const double *source,
                       ptrdiff_t tst, ptrdiff_t sst, size_t len,
                       int pe);
void shmem_float_iget(float *target, const float *source,
                      ptrdiff_t tst, ptrdiff_t sst, size_t len,
                      int pe);
void shmem_iget32(void *target, const void *source, ptrdiff_t tst,
                  ptrdiff_t sst, size_t len, int pe);
void shmem_iget64(void *target, const void *source, ptrdiff_t tst,
                  ptrdiff_t sst, size_t len, int pe);
void shmem_iget128(void *target, const void *source,
                   ptrdiff_t tst, ptrdiff_t sst, size_t len,
                   int pe);
void shmem_int_iget(int *target, const int *source, ptrdiff_t tst,
                    ptrdiff_t sst, size_t len, int pe);
void shmem_long_iget(long *target, const long *source,
                     ptrdiff_t tst, ptrdiff_t sst, size_t len,
                     int pe);
void shmem_longdouble_iget(long double *target,
                           const long double *source,
                           ptrdiff_t tst, ptrdiff_t sst,
                           size_t len, int pe);
void shmem_longlong_iget(long long *target,
                         const long long *source, ptrdiff_t tst,
                         ptrdiff_t sst, size_t len, int pe);
void shmem_short_iget(short *target, const short *source,
                      ptrdiff_t tst, ptrdiff_t sst, size_t len,
                      int pe);
```

PARAMETERS

target	Array to be updated on the local PE.

`source`	Array containing the data to be copied on the remote PE.
`tst`	The stride between consecutive elements of the `target` array. The stride is scaled by the element size of the `target` array. A value of `1` indicates contiguous data.
`sst`	The stride between consecutive elements of the `source` array. The stride is scaled by the element size of the `source` array. A value of `1` indicates contiguous data.
`len`	Number of elements in the `target` and `source` arrays.
`pe`	The number of the remote PE on which `source` resides.

DESCRIPTION

These routines provide the means for copying a strided array from a remote PE to a local strided array. The routines return when the data has been copied into the local `target` array.

The function `shmem_iget()` reads strided array where each element is any non-character type that has a storage size equal to 64 bits from the remote PE.

The function `shmem_double_iget()` reads strided array of type `double` from the remote PE.

The function `shmem_float_iget()` reads strided array of type `float` from the remote PE.

The function `shmem_iget32()` reads any non-character type that has a storage size equal to 32 bits from the remote PE.

The function `shmem_iget64()` reads strided array where each element is any non-character type that has a storage size equal to 64 bits from the remote PE.

The function `shmem_iget128()` reads strided array where each element is any non-character type that has a storage size equal to 128 bits from the remote PE.

The function `shmem_int_iget()` reads strided array of type `integer` from the remote PE.

The function `shmem_long_iget()` reads strided array of type `long` from the remote PE.

The function `shmem_longdouble_iget()` reads strided array of type `long double` from the remote PE.

The function `shmem_longlong_iget()` reads strided array of type `long long` from the remote PE.

The function `shmem_short_iget()` reads strided array of type `short` from the remote PE.

SEE ALSO

`shmem_iput``(3)`, **`shmem_put`**`(3)`, **`shmem_get`**`(3)`, **`shmem_quiet`**`(3)`

2.8 Synchronization Operations

The synchronisation functions are listed in Table 2.5.

Table 2.5: Synchronization Functions

Name	Description
shmem_barrier	Performs a barrier operation on a subset of PEs
barrier	Performs a barrier operation on all PEs
shmem_barrier_all	Performs a barrier operation on all PEs
shmem_int_wait	Waits for an integer variable to change on the local PE
shmem_int_wait_until	Waits for an integer variable to change and satisfy a condition
shmem_long_wait	Waits for a long variable to change on the local PE
shmem_long_wait_until	Waits for a long variable to change and satisfy a condition
shmem_longlong_wait	Waits for a long variable to change on the local PE
shmem_longlong_wait_until	Waits for a long long variable to satisfy a condition
shmem_short_wait	Waits for a short variable to change on the local PE
shmem_short_wait_until	Waits for a short variable to change and satisfy a condition
shmem_wait	Waits for a long variable to change on the local PE
shmem_short_wait	Waits for a long variable to change and satisfy a condition
shmem_fence	Assures ordering of delivery of puts
shmem_quiet	Waits for completion of all outstanding remote writes

These functions are used to express different kinds of synchronization. The routines barrier, shmem_barrier and shmem_barrier_all_barrier are used to synchronize all or a subset of the processes belonging to the parallel application.

The routines like shmem_wait are used to synchronize a pair of processing elements. The PE calling one of these functions on a local variable *V* is blocked until a remote PE changes the value of *V*.

The function shmem_fence ensures ordering of remote write (put) operations. All put operations issued to a particular processing element (PE) prior to the call to shmem_fence are guaranteed to be delivered before any subsequent put operations to the same PE which follow the call to shmem_fence.

The function shmem_quiet waits for completion of all outstanding remote writes initiated from the current PE. The routine shmem_quiet does not return until all data is delivered to the remote PEs memory.

The synchronization functions are described in detail on the following pages.

NAME

barrier, shmem_barrier_all – register the arrival of a PE at a barrier and suspends PE execution until all other PE arrive at the barrier

SYNOPSIS

```
#include <shmem.h>

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

DESCRIPTION

Barriers are a fast mechanism for synchronizing all PEs at once.

The function `shmem_barrier_all()` cause a PE to suspend execution until all PEs have called `shmem_barrier_all()`. These barrier functions also ensure completion of all previously issued local memory stores and remote memory updates issued via shared memory routine calls such as shmem_put32().

SEE ALSO

shmem_barrier(3), **shmem_init**(3)

NAME

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

SYNOPSIS

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

PARAMETERS

`PE_start`	The lowest virtual PE number of the active set of PEs. `PE_start` must be of type integer. If you are using Fortran, it must be a default integer value.
`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE numbers in the active set.
`PE_size`	The number of PEs in the active set. `PE_size` must be of type integer.
`pSync`	A symmetric work array. `pSync` must have size `_SHMEM_BARRIER_SYNC_SIZE`. Every element of this array must be initialized to 0 before any of the PEs in the active set enter `shmem_barrier` the first time.

DESCRIPTION

The `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 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. The `shmem_barrier` routine ensures that all previously issued local stores and previously issued remote memory updates done by any of the PEs in the active set (by using Shmem calls, for example `shmem_put`) are complete before returning. The same `pSync` array may be reused on consecutive calls to `shmem_barrier` if the same active PE set is used.

SEE ALSO

`shmem_barrier_all(3)`,

NAME

shmem_wait, shmem_int_wait, shmem_long_wait, shmem_longlong_wait, shmem_short_wait, shmem_wait_until, shmem_int_wait_until, shmem_long_wait_until, shmem_longlong_wait_until, shmem_short_wait_until – Waits for a variable on the local processing element (PE) to change

SYNOPSIS

```
#include <shmem.h>
void shmem_wait(long *var, long value);
void shmem_int_wait(int *var, int value);
void shmem_long_wait(long *var, long value);
void shmem_longlong_wait(long long *var, long long value);
void shmem_short_wait(short *var, short value);
void shmem_wait_until(long *var, int cond, long value);
void shmem_int_wait_until(int *var, int cond, int value);
void shmem_long_wait_until(long *var, int cond, long value);
void shmem_longlong_wait_until(long long *var, int cond,
                               long long value);
void shmem_short_wait_until(short *var, int cond, short value);
```

PARAMETERS

`var` — A remotely accessible integer variable that is being updated by a remote processing element.

`cond` — The compare operator that compares `var` with `value`. The following `cond` values are supported:

`SHMEM_CMP_EQ`	*Equal* operator
`SHMEM_CMP_NE`	*Not equal* operator
`SHMEM_CMP_GT`	*Greater then* operator
`SHMEM_CMP_LE`	*Less then or equal* operator
`SHMEM_CMP_LT`	*Less then operator* operator
`SHMEM_CMP_GE`	*Greater then or equal* operator

`value` — Is the value used as right operand of the compare operator `cond`. The left one is the value pointed by `var`

DESCRIPTION

These functions wait for `var` to be changed by a write (put) or atomic swap issued by a remote PE. These routines can be used for point-to-point direct synchronization and

offer a mechanism to notify a PE that another process element has completed some action.

The function `shmem_wait()` blocks the calling PE until some remote PE writes a `long` value, not equal to `value`, into `var` on the waiting PE.

The function `shmem_int_wait()` blocks the calling PE until some remote PE writes an `integer` value, not equal to `value`, into `var` on the waiting PE.

The function `shmem_long_wait()` blocks the calling PE until some remote PE writes a `long` value, not equal to `value`, into `var` on the waiting PE.

The function `shmem_longlong_wait()` blocks the calling PE until some remote PE writes a `long long` value, not equal to `value`, into `var` on the waiting PE.

The function `shmem_short_wait()` blocks the calling PE until some remote PE writes a `short` value, not equal to `value`, into `var` on the waiting PE.

The function `shmem_wait_until()` blocks the calling PE until some remote PE changes the `long` variable `var` to satisfy the condition implied by `comp` and `val`.

The function `shmem_int_wait_until()` blocks the calling PE until some remote PE changes the `integer` variable `var` to satisfy the condition implied by `comp` and `val`.

The function `shmem_long_wait_until()` blocks the calling PE until some remote PE changes the `long` variable `var` to satisfy the condition implied by `comp` and `val`.

The function `shmem_longlong_wait_until()` blocks the calling PE until some remote PE changes the `long long` variable `var` to satisfy the condition implied by `comp` and `val`.

The function `shmem_short_wait_until()` blocks the calling PE until some remote PE changes the `short` variable `var` to satisfy the condition implied by `comp` and `val`.

SEE ALSO

shmem_put(3)

NAME

shmem_fence – assures ordering of delivery of puts

SYNOPSIS

```
#include <shmem.h>
void shmem_fence(void);
```

DESCRIPTION

This function ensures ordering of remote write (put) operations. All put operations issued to a particular processing element (PE) prior to the call to `shmem_fence` are guaranteed to be delivered before any subsequent remote write operation to the same PE which follows the call to `shmem_fence`. The `shmem_quiet` function should be called if ordering of puts is desired when multiple remote PEs are involved.

SEE ALSO

shmem_quiet(3)

NAME

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

SYNOPSIS

```
#include <shmem.h>
void shmem_quiet(void);
```

DESCRIPTION

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

SEE ALSO

shmem_put(3), **shmem_fence**(3), **shmem_barrier**(3), **shmem_wait**(3)

2.9 Atomic Memory Operations

The atomic memory functions are listed in Table 2.6.

Table 2.6: Atomic Memory Operations

Name	Description
shmem_double_swap	Atomic swap to a remote double data object
shmem_float_swap	Atomic swap to a remote float data object
shmem_int_swap	Atomic swap to a remote integer data object
shmem_long_swap	Atomic swap to a remote long data object
shmem_longlong_swap	Atomic swap to a remote long long data object
shmem_short_swap	Atomic swap to a remote short data object
shmem_swap	Atomic swap to a remote long data object
shmem_int_cswap	Atomic conditional swap to a remote integer data object
shmem_long_cswap	Atomic conditional swap to a remote long data object
shmem_longlong_cswap	Atomic conditional swap to a remote long long data object
shmem_short_cswap	Atomic conditional swap to a remote short data object
shmem_short_add	Atomic add on remote short data object
shmem_int_mswap	Atomic masked swap to an integer data object
shmem_long_mswap	Atomic masked swap to an long data object
shmem_short_mswap	Atomic masked swap to a short data object
shmem_int_fadd	Atomic fetch-and-add on an integer data object
shmem_long_fadd	Atomic fetch-and-add on a long data object
shmem_longlong_fadd	Atomic fetch-and-add on a longlong data object
shmem_short_fadd	Atomic fetch-and-add on a short data object
shmem_int_finc	Atomic fetch-and-increment on an integer data object
shmem_long_finc	Atomic fetch-and-increment on a long data object
shmem_longlong_finc	Atomic fetch-and-increment on a longlong data object
shmem_short_finc	Atomic fetch-and-increment on a short data object
shmem_short_inc	Atomic increment on a short data object

These routines are used to perform atomic read-and-update operations on a remote data object. It is worth noting that the atomicty accessing a shared variable *V* is only guaranteed if *V* is updated using the Shmem routines only. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which *V* is local, must refer to *V* using the Shmem atomic routines.

Routines like shmem_swap, shmem_int_cswap and shmem_int_mswap perform an atomic swap operation, an atomic conditional swap operation and a masked atomic swap operation to a remote data object respectively. The functions like shmem_int_fadd, shmem_int_finc and shmem_short_add perform atomic fetch-and-add, fetch-and-increment and atomic add on a remote data object respectively.

The functions performing atomic memory operations are described in detail on the following pages.

NAME

shmem_swap, shmem_double_swap, shmem_float_swap, shmem_int_swap, shmem_long_swap, shmem_longlong_swap, shmem_short_swap – Perform an atomic swap to a remote data object

SYNOPSIS

```
#include <shmem.h>

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

PARAMETERS

`target`	The pointer to the remotely accessible data object to be updated on the remote PE. The type of `target` should match that implied in the SYNOPSIS section.
`value`	Value to be atomically written to the remote PE. `value` is the same type as `target`.
`pe`	An integer indicating the PE number on which `target` is to be updated.

DESCRIPTION

These functions perform atomic swap operations.It is worth noting that the atomic access to a variable *V* is only guaranteed if *V* is updated solely by Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which the variable *V* is local, must refer *V* using the Shmem atomic routines.

The `shmem_swap` function writes the `long` value `value` in to the variable pointed by `target` on processing element `pe` and returns the previous contents of `target` as an atomic operation.

The `shmem_double_swap` function writes the `double` value `value` in to the variable pointed by `target` on processing element `pe` and returns the previous contents of `target` as an atomic operation.

The shmem_float_swap function writes the float value value in to the variable pointed by target on processing element pe and returns the previous contents of target as an atomic operation.

The shmem_int_swap function writes the integer value value in to the variable pointed by target on processing element pe and returns the previous contents of target as an atomic operation.

The shmem_long_swap function writes the long value value in to the variable pointed by target on processing element pe and returns the previous contents of target as an atomic operation.

The shmem_longlong_swap function writes the longlong value value in to the variable pointed by target on processing element pe and returns the previous contents of target as an atomic operation.

The shmem_short_swap function writes the short value value in to the variable pointed by target on processing element pe and returns the previous contents of target as an atomic operation.

RETURN VALUES

These functions return the contents that had been at the target address on the remote PE prior to the swap is returned.

SEE ALSO

shmem_put(3)

NAME

shmem_int_cswap, shmem_long_cswap, shmem_longlong_cswap, shmem_short_cswap – Performs an atomic conditional swap to a remote data object

SYNOPSIS

```
#include <shmem.h>

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

PARAMETERS

`target`	The pointer to a remotely accessible data object to be updated on the remote PE. The data type of `target` should match that implied in the SYNOPSIS section.
`cond`	The value of `cond` is compared to the remote `target` value. If `cond` and the remote target value are equal then `value` is swapped in the remote target. Otherwise the remote target is unchanged. In either case, the old value of the remote target is returned as the function return value. The parameter `cond` must be of the same data type of `target`.
`value`	The value to be atomically written to the remote PE. `value` must be the same data type as `target`.
`pe`	An integer that indicates the PE number upon which `target` is to be updated.

DESCRIPTION

The conditional swap routines conditionally update a target data object on an arbitrary processing element (PE) and return prior contents of the data object in one atomic operation. It is worth noting that atomic access to a variable *V* is only guaranteed if *V* is updated solely by Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which the variable *V* is local, must refer to *V* using the Shmem atomic routines.

The function `shmem_int_cswap` performs an atomic conditional operation on a remotely accessible `integer` data object.

The function `shmem_long_cswap` performs an atomic conditional operation on a remotely accessible `long` data object.

The function `shmem_longlong_cswap` performs an atomic conditional operation on a remotely accessible `long long` data object.

The function `shmem_short_cswap` performs an atomic conditional operation on remotely accessible `short` data object.

RETURN VALUES

These functions return the contents that had been at the target address on the remote PE prior to the conditional swap.

SEE ALSO

shmem_swap(3)

NAME

shmem_short_add – performs an atomic add operation on a remote data object

SYNOPSIS

```
#include <shmem.h>
void shmem_short_add(short *target, short value, int pe);
```

PARAMETERS

`target`	The pointer to a remotely accessible data object to be updated on the remote PE. The data type of `target` should match that implied in the SYNOPSIS section.
`value`	The value to be atomically added to the `target`.
`pe`	An integer that indicates the PE number upon which `target` is to be updated.

DESCRIPTION

The `shmem_short_add` routine performs an atomic add operation. It adds `value` to the variable pointed by `target` on the processing element specified by `pe`. It is worth noting that the atomic access to a variable *V* is only guaranteed if *V* is updated solely by Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which the variable *V* is local, must refer to *V* using Shmem atomic routines.

SEE ALSO

shmem_short_cswap(3)

NAME

shmem_int_mswap, shmem_long_mswap, shmem_short_mswap – perform an atomic masked swap on a remote data object

SYNOPSIS

```
#include <shmem.h>

int shmem_int_mswap(int *target, int mask, int value, int pe);
long shmem_long_mswap(long *target, long mask, long value,
                      int pe);
short shmem_short_mswap(short *target, short mask, short value,
                        int pe);
```

PARAMETERS

`target`	The pointer to a remotely accessible data object to be updated on the remote PE. The data type of `target` should match that implied in the SYNOPSIS section.
`mask`	Identifies the bits within `target` that are to be updated with bits from `value`. The bits set to 1 in `mask` indicate bits to be copied from `value` into the corresponding bit location in `target`. The parameter `mask` must be the same data type as `target`.
`value`	Contains the bits to be atomically written to `target` on the remote PE. The parameter `mask` identifies the bits to be transferred. The parameter `value` must be the same data type as `target`.
`pe`	An integer that indicates the PE number upon which `target` is to be updated.

DESCRIPTION

The masked swap routines update a target data object on an arbitrary processing element (PE) and return the prior content of the data object in one atomic operation. It is worth noting that atomic access to a variable *V* is only guaranteed if *V* is updated solely by this Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which *V* is local, must refer to the variable *V* using Shmem atomic routines.

The `shmem_int_mswap` routine updates atomically the `integer` value pointed by `target` according to the bit mask specified by `mask`.

The `shmem_long_mswap` routine updates atomically the `long` value pointed by `target` according to the bit mask specified by `mask`.

The shmem_short_mswap routine updates atomically the short value pointed by target according to the bit mask specified by mask.

RETURN VALUES

These functions return the contents that had been in the target address on the remote PE prior to the masked swap.

SEE ALSO

shmem_int_swap(3), **shmem_int_cswap**(3)

NAME

shmem_int_fadd, shmem_long_fadd, shmem_longlong_fadd, shmem_short_fadd – perform an atomic fetch-and-add operation on a remote data object

SYNOPSIS

```
#include <shmem.h>

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

PARAMETERS

`target`	The pointer to a remotely accessible data object to be updated on the remote PE. The data type of `target` should match that implied in the SYNOPSIS section.
`value`	The value to be atomically added to `target`. The type of `value` should match that implied in the SYNOPSIS section.
`pe`	An integer that indicates the PE number upon which `target` is to be updated.

DESCRIPTION

These routines perform an atomic fetch-and-add operation adding `value` to `target` on PE specified by `pe` and returning the previous contents of the `target`. It is worth noting that the atomic access a variable *V* is only guaranteed if *V* is updated solely by this Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which the variable *V* is local, must refer to *V* using the Shmem atomic routines.

The `shmem_int_fadd` operates on `integer` data object.

The `shmem_long_fadd` operates on `long` data object.

The `shmem_longlong_fadd` operates on `long long` data object.

The `shmem_longshort_fadd` operates on `short` data object.

RETURN VALUES

These functions return the contents that had been at the `target` address on the remote PE prior to the atomic addition operation.

SEE ALSO

shmem_int_swap(3), **shmem_int_cswap**(3) **shmem_int_finc**(3)

NAME

shmem_int_finc, shmem_long_finc, shmem_longlong_finc, shmem_short_finc – perform an atomic fetch-and-increment operation on a remote data object

SYNOPSIS

```
#include <shmem.h>

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

PARAMETERS

`target`	The pointer to a remotely accessible data object to be incremented on the remote PE. The data type of `target` should match that implied in the SYNOPSIS section.
`pe`	An integer that indicates the PE number upon which `target` is to be updated.

DESCRIPTION

These routines perform an atomic fetch-and-increment operation. They increment the data objet pointed by `target` on PE specified by `pe` and return the previous contents of `target` as an atomic operation. It is worth noting that the atomic access to a variable *V* is only guaranteed if *V* is updated solely by this Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which the variable *V* is local, must refer to *V* using the Shmem atomic routines.

The `shmem_int_finc` operates on `integer` data object.

The `shmem_long_finc` operates on `long` data object.

The `shmem_longlong_finc` operates on `long long` data object.

The `shmem_longshort_finc` operates on `short` data object.

RETURN VALUES

These functions return the contents that had been at the `target` address on the remote PE prior to the atomic increment.

SEE ALSO

shmem_int_swap(3), **shmem_int_cswap**(3) **shmem_int_fadd**(3)

NAME

shmem_short_inc – perform an atomic increment operation on a remote data object

SYNOPSIS

```
#include <shmem.h>
void shmem_short_inc(short *target, int pe);
```

PARAMETERS

`target`	The pointer to a remotely accessible data object to be incremented on the remote PE.
`pe`	An integer that indicates the PE number upon which `target` is to be updated.

DESCRIPTION

This routine performs an atomic increment on a remote variable pointed by `target` on PE specified by `pe`. It is worth noting that the atomic access to a variable *V* is only guaranteed if *V* is updated solely by this Shmem routines. Thus, in order to preserve the correct semantic of atomic operations all the processing elements, including the one for which the variable *V* is local, must refer to *V* using the Shmem atomic routines.

SEE ALSO

shmem_short_swap(3), **shmem_short_finc**(3) **shmem_short_fadd**(3)

2.10 Collective Reduction Operations

The collective reduction functions are listed in Table 2.7.

Table 2.7: Collective Reduction Operation

Name	Description
shmem_int_and_to_all	Performs a logical AND function on integer
shmem_long_and_to_all	Performs a logical AND reduction on long
shmem_longlong_and_to_all	Performs a logical AND reduction on long long
shmem_short_and_to_all	Performs a logical AND reduction on short
shmem_double_max_to_all	Performs a maximum function on double
shmem_float_max_to_all	Performs a maximum function on float
shmem_int_max_to_all	Performs a maximum function on int
shmem_long_max_to_all	Performs a maximum function on long
shmem_longdouble_max_to_all	Performs a maximum function on long double
shmem_longlong_max_to_all	Performs a maximum function on array of long long
shmem_short_max_to_all	Performs a maximum function of short
shmem_double_min_to_all	Performs a minimum function on array of double
shmem_float_min_to_all	Performs a minimum function on float
shmem_int_min_to_all	Performs a minimum function on int
shmem_long_min_to_all	Performs a minimum function on long
shmem_longdouble_min_to_all	Performs a minimum function on long double
shmem_longlong_min_to_all	Performs a minimum function on long long
shmem_short_min_to_all	Performs a minimum function on short
shmem_int_or_to_all	Performs a logical OR function on integer
shmem_long_or_to_all	Performs a logical OR function on long
shmem_longlong_or_to_all	Performs a logical OR function on long long
shmem_short_or_to_all	Performs a logical OR function on short
shmem_double_prod_to_all	Performs a product reduction on double
shmem_float_prod_to_all	Performs a product reduction on float
shmem_int_prod_to_all	Performs a product reduction on int
shmem_long_prod_to_all	Performs a product reduction on long
shmem_longdouble_prod_to_all	Performs a product reduction on long double
shmem_longlong_prod_to_all	Performs a product reduction on long long
shmem_short_prod_to_all	Performs a product reduction on short
shmem_double_sum_to_all	Performs a sum reduction on double
shmem_float_sum_to_all	Performs a sum reduction on float
shmem_int_sum_to_all	Performs a sum reduction on int
shmem_long_sum_to_all	Performs a sum reduction on long
shmem_longdouble_sum_to_all	Performs a sum reduction on long double
shmem_longlong_sum_to_all	Performs a sum reduction on long long
shmem_short_sum_to_all	Performs a sum reduction on short
shmem_int_xor_to_all	Performs a logical exclusive OR on short
shmem_long_xor_to_all	Performs a logical exclusive OR on array of long
shmem_longlong_xor_to_all	Performs a logical exclusive OR on long long

(continued on next page)

Table 2.7: Collective Reduction Operation (cont.)

Name	Description
`shmem_short_xor_to_all`	Performs a logical exclusive OR on `short`

The Shmem library supplies a wide number of functions to perform associative binary operations across a set of values distributed on a set of processing elements. The following associative binary operators are supported:

`AND` The logical AND function (e.g. `shmem_int_and_all_to_all`)

`MAX` The *maximum* function (e.g. `shmem_int_max_all_to_all`)

`MIN` The *minimum* function (e.g. `shmem_int_min_all_to_all`)

`OR` The logical OR function (e.g. `shmem_int_or_all_to_all`)

`PROD` The *product* function (e.g. `shmem_int_prod_all_to_all`)

`SUM` The *sum* function (e.g. `shmem_int_sum_all_to_all`)

`XOR` The logical exclusive OR function (e.g. `shmem_int_xor_all_to_all`)

The collective reduction functions are described in detail on the following pages.

NAME

shmem_int_and_to_all, shmem_long_and_to_all, shmem_longlong_and_to_all, shmem_short_and_to_all – perform a logical AND function across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
void shmem_int_and_to_all(int *target, int *source, int nreduce,
                          int PE_start, int logPE_stride,
                          int PE_size, int *pWrk, long *pSync);
void shmem_long_and_to_all(long *target, long *source,
                           int nreduce, int PE_start,
                           int logPE_stride, int PE_size,
                           long *pWrk, long *pSync);
void shmem_longlong_and_to_all(long long *target,
                               long long *source, int nreduce,
                               int PE_start, int logPE_stride,
                               int PE_size, long long *pWrk,
                               long *pSync);
void shmem_short_and_to_all(short *target, short *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            short *pWrk, long *pSync);
```

PARAMETERS

`target`	A symmetric array, of length `nreduce`, to receive the result of the reduction operation. The data type of `target` should match that implied in the SYNOPSIS section.
`source`	A symmetric array, of length `nreduce`, that contains one element for each separete reduction operation. The `source` argument must have the same data type as `target`.
`nreduce`	The number of elements in the `target` and `source` array.
`PE_start`	The lowest virtual PE number of the active set of PEs.
`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE number in the active set.
`PE_size`	The number of PEs in the active set.
`pWrk`	A symmetric work array. The `pWrk` argument must have the same data type as `target`. In C/C++, this contains `max(nreduce/2+1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync	A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The nreduce argument determines the number of elements to perform the reduction operation on. 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 values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pWrk and pSync arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The target array on all PEs in the active set is ready to accept the results of the reduction.

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

- The target array is updated.
- The values in the pSync array are restored to the original values.

The function shmem_int_and_to_all performs a reduction applaying the logical AND operator to integer values distributed across the PEs.

The function shmem_long_and_to_all performs a reduction applaying the logical AND operator to long values distributed across the PEs.

The function shmem_longlong_and_to_all performs a reduction applaying the logical AND operator to long long values distributed across the PEs.

The function shmem_short_and_to_all performs a reduction applaying the logical AND operator to short values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_double_max_to_all, shmem_float_max_to_all, shmem_int_max_to_all, shmem_long_max_to_all, shmem_longdouble_max_to_all, shmem_longlong_max_to_all, shmem_short_max_to_all – performs a maximum function reduction across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
void shmem_double_max_to_all(double *target, double *source,
                             int nreduce, int PE_start,
                             int logPE_stride, int PE_size,
                             double *pWrk, long *pSync);
void shmem_float_max_to_all(float *target, float *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            float *pWrk, long *pSync);
void shmem_int_max_to_all(int *target, int *source, int nreduce,
                          int PE_start, int logPE_stride,
                          int PE_size, int *pWrk, long *pSync);
void shmem_long_max_to_all(long *target, long *source,
                           int nreduce, int PE_start,
                           int logPE_stride, int PE_size,
                           long *pWrk, long *pSync);
void shmem_longdouble_max_to_all(long double *target,
                                 long double *source, int nreduce,
                                 int PE_start, int logPE_stride,
                                 int PE_size, long double *pWrk,
                                 long *pSync);
void shmem_longlong_max_to_all(long long *target,
                               long long *source, int nreduce,
                               int PE_start, int logPE_stride,
                               int PE_size, long long *pWrk,
                               long *pSync);
void shmem_short_max_to_all(short *target, short *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            short *pWrk, long *pSync);
```

PARAMETERS

`target` A symmetric array, of length `nreduce`, to receive the result of the reduction operations. The data type of `target` should match that

	implied in the SYNOPSIS section.
source	A symmetric array, of length nreduce, that contains one element for each separete reduction operation. The source argument must have the same data type as target.
nreduce	The number of elements in the target and source array.
PE_start	The lowest virtual PE number of the active set of PEs.
logPE_stride	The log (base 2) of the stride between consecutive virtual PE number in the active set. of type integer.
PE_size	The number of PEs in the active set.
pWrk	A symmetric work array. The pWrk argument must have the same data type as target. In C/C++, this contains max(nreduce/2+1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements.
pSync	A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The nreduce argument determines the number of elements to perform the reduction operation on. 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 values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pWrk and pSync arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The target array on all PEs in the active set is ready to accept the results of the reduction.

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

- The target array is updated.
- The values in the pSync array are restored to the original values.

The function `shmem_double_max_to_all` performs a reduction applaying the maximum function to `doubles` values distributed across the PEs.

The function `shmem_float_max_to_all` performs a reduction applaying the maximum function to `float` values distributed across the PEs.

The function `shmem_int_max_to_all` performs a reduction applaying the maximum function to `integer` values distributed across the PEs.

The function `shmem_long_max_to_all` performs a reduction applaying the maximum function to `long` values distributed across the PEs.

The function `shmem_longdouble_max_to_all` performs a reduction applaying the maximum function to `long double` values distributed across the PEs.

The function `shmem_longlong_max_to_all` performs a reduction applaying the maximum function to `long long` values distributed across the PEs.

The function `shmem_short_max_to_all` performs a reduction applaying the maximum function to `short` values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_double_min_to_all, shmem_float_min_to_all, shmem_int_min_to_all, shmem_long_min_to_all, shmem_longdouble_min_to_all, shmem_longlong_min_to_all, shmem_short_min_to_all – performs a minimum function reduction across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
void shmem_double_min_to_all(double *target, double *source,
                             int nreduce, int PE_start,
                             int logPE_stride, int PE_size,
                             double *pWrk, long *pSync);
void shmem_float_min_to_all(float *target, float *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            float *pWrk, long *pSync);
void shmem_int_min_to_all(int *target, int *source, int nreduce,
                          int PE_start, int logPE_stride,
                          int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *target, long *source,
                           int nreduce, int PE_start,
                           int logPE_stride, int PE_size,
                           long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *target,
                                 long double *source, int nreduce,
                                 int PE_start, int logPE_stride,
                                 int PE_size, long double *pWrk,
                                 long *pSync);
void shmem_longlong_min_to_all(long long *target,
                               long long *source, int nreduce,
                               int PE_start, int logPE_stride,
                               int PE_size, long long *pWrk,
                               long *pSync);
void shmem_short_min_to_all(short *target, short *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            short *pWrk, long *pSync);
```

PARAMETERS

`target` A symmetric array, of length `nreduce`, to receive the result of the reduction operations. The data type of `target` should match that

implied in the SYNOPSIS section.

source	A symmetric array, of length nreduce, that contains one element for each separete reduction operation. The source argument must have the same data type as target.
nreduce	The number of elements in the target and source array
PE_start	The lowest virtual PE number of the active set of PEs.
logPE_stride	The log (base 2) of the stride between consecutive virtual PE number in the active set.
PE_size	The number of PEs in the active set.
pWrk	A symmetric work array. The pWrk argument must have the same data type as target. In C/C++, this contains max(nreduce/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements.
pSync	A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The nreduce argument determines the number of elements to perform the reduction operation on. 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 values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pWrk and pSync arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The target array on all PEs in the active set is ready to accept the results of the reduction.

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

- The target array is updated.
- The values in the pSync array are restored to the original values.

The function `shmem_double_min_to_all` performs a reduction applaying the minimum function to `doubles` values distributed across the PEs.

The function `shmem_float_min_to_all` performs a reduction applaying the minimum function to `float` values distributed across the PEs.

The function `shmem_int_min_to_all` performs a reduction applaying the minimum function to `integer` values distributed across the PEs.

The function `shmem_long_min_to_all` performs a reduction applaying the minimum function to `long` values distributed across the PEs.

The function `shmem_longdouble_min_to_all` performs a reduction applaying the minimum function to `long double` values distributed across the PEs.

The function `shmem_longlong_min_to_all` performs a reduction applaying the minimum function to `long long` values distributed across the PEs.

The function `shmem_short_min_to_all` performs a reduction applaying the minimum function to `short` values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_int_or_to_all, shmem_long_or_to_all, shmem_longlong_or_to_all, shmem_short_or_to_all – perform a logical OR function across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
void shmem_int_or_to_all(int *target, int *source, int nreduce,
                         int PE_start, int logPE_stride,
                         int PE_size, int *pWrk, long *pSync);
void shmem_long_or_to_all(long *target, long *source, int nreduce,
                          int PE_start, int logPE_stride,
                          int PE_size, long *pWrk, long *pSync);
void shmem_longlong_or_to_all(long long *target,
                              long long *source, int nreduce,
                              int PE_start, int logPE_stride,
                              int PE_size, long long *pWrk,
                              long *pSync);
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);
```

PARAMETERS

`target`	A symmetric array, of length `nreduce`, to receive the result of the reduction operation. The data type of `target` should match that implied in the SYNOPSIS section.
`source`	A symmetric array, of length `nreduce`, that contains one element for each separete reduction operation. The `source` argument must have the same data type as `target`.
`nreduce`	The number of elements in the `target` and `source` array.
`PE_start`	The lowest virtual PE number of the active set of PEs.
`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE number in the active set.
`PE_size`	The number of PEs in the active set.
`pWrk`	A symmetric work array. The `pWrk` argument must have the same data type as `target`. In C/C++, this contains `max(nreduce/2+1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync	A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The nreduce argument determines the number of elements to perform the reduction operation on. 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 values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pWrk and pSync arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The target array on all PEs in the active set is ready to accept the results of the reduction.

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

- The target array is updated.
- The values in the pSync array are restored to the original values.

The function shmem_int_or_to_all performs a reduction applaying the logical OR operator on integer values distributed across the PEs.

The function shmem_long_or_to_all performs a reduction applaying the logical OR operator on long values distributed across the PEs.

The function shmem_longlong_or_to_all performs a reduction applaying the logical OR operator on long long values distributed across the PEs.

The function shmem_short_or_to_all performs a reduction applaying the logical OR operator on short values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_double_prod_to_all, shmem_float_prod_to_all, shmem_int_prod_to_all, shmem_long_prod_to_all, shmem_longdouble_prod_to_all, shmem_longlong_prod_to_all, shmem_short_prod_to_all – performs a product reduction across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
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_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_int_prod_to_all(int *target, int *source, int nreduce,
                           int PE_start, int logPE_stride,
                           int PE_size, int *pWrk, long *pSync);
void shmem_long_prod_to_all(long *target, long *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            long *pWrk, long *pSync);
void shmem_longdouble_prod_to_all(long double *target,
                                  long double *source,
                                  int nreduce, int PE_start,
                                  int logPE_stride, int PE_size,
                                  long double *pWrk, long *pSync);
void shmem_longlong_prod_to_all(long long *target,
                                long long *source, int nreduce,
                                int PE_start, int logPE_stride,
                                int PE_size, long long *pWrk,
                                long *pSync);
void shmem_short_prod_to_all(short *target, short *source,
                             int nreduce, int PE_start,
                             int logPE_stride, int PE_size,
                             short *pWrk, long *pSync);
```

PARAMETERS

`target`	A symmetric array, of length `nreduce`, to receive the result of the reduction operations. The data type of `target` should match that implied in the SYNOPSIS section.
`source`	A symmetric array, of length `nreduce`, that contains one element for each separete reduction operation. The `source` argument must have the same data type as `target`.
`nreduce`	The number of elements in the `target` and `source` array.
`PE_start`	The lowest virtual PE number of the active set of PEs.
`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE number in the active set.
`PE_size`	The number of PEs in the active set.
`pWrk`	A symmetric work array. The `pWrk` argument must have the same data type as `target`. In C/C++, this contains `max(nreduce/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.
`pSync`	A symmetric work array. In C/C++, `pSync` must be of type `long` and size `_SHMEM_REDUCE_SYNC_SIZE`. Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The `nreduce` argument determines the number of separate reduction 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 values of arguments `nreduce`, `PE_start`, `logPE_stride`, and `PE_size` must be equal on all PEs in the active set. The same `target` and `source` arrays, and the same `pWrk` and `pSync` work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The `pWrk` and `pSync` arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The `target` array on all PEs in the active set is ready to accept the results of the reduction.

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

- The `target` array is updated.
- The values in the `pSync` array are restored to the original values.

The function `shmem_double_prod_to_all` performs a reduction applaying the product function to `doubles` values distributed across the PEs.

The function `shmem_float_prod_to_all` performs a reduction applaying the product function to `float` values distributed across the PEs.

The function `shmem_int_prod_to_alql` performs a reduction applaying the product function to `integer` values distributed across the PEs.

The function `shmem_long_prod_to_all` performs a reduction applaying the product function to `long` values distributed across the PEs.

The function `shmem_longdouble_prod_to_all` performs a reduction applaying the product function to `long double` values distributed across the PEs.

The function `shmem_longlong_prod_to_all` performs a reduction applaying the product function to `long long` values distributed across the PEs.

The function `shmem_short_prod_to_all` performs a reduction applaying the product function to `short` values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_double_sum_to_all, shmem_float_sum_to_all, shmem_int_sum_to_all, shmem_long_sum_to_all, shmem_longdouble_sum_to_all, shmem_longlong_sum_to_all, shmem_short_sum_to_all – performs a product reduction across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
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_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_int_sum_to_all(int *target, int *source, int nreduce,
                          int PE_start, int logPE_stride,
                          int PE_size, int *pWrk, long *pSync);
void shmem_long_sum_to_all(long *target, long *source,
                           int nreduce, int PE_start,
                           int logPE_stride, int PE_size,
                           long *pWrk, long *pSync);
void shmem_longdouble_sum_to_all(long double *target,
                                 long double *source, int nreduce,
                                 int PE_start, int logPE_stride,
                                 int PE_size, long double *pWrk,
                                 long *pSync);
void shmem_longlong_sum_to_all(long long *target,
                               long long *source, int nreduce,
                               int PE_start, int logPE_stride,
                               int PE_size, long long *pWrk,
                               long *pSync);
void shmem_short_sum_to_all(short *target, short *source,
                            int nreduce, int PE_start,
                            int logPE_stride, int PE_size,
                            short *pWrk, long *pSync);
```

PARAMETERS

`target` A symmetric array, of length `nreduce`, to receive the result of the reduction operations. The data type of `target` should match that

implied in the SYNOPSIS section.

source	A symmetric array, of length nreduce, that contains one element for each separete reduction operation. The source argument must have the same data type as target.
nreduce	The number of elements in the target and source array.
PE_start	The lowest virtual PE number of the active set of PEs.
logPE_stride	The log (base 2) of the stride between consecutive virtual PE number in the active set.
PE_size	The number of PEs in the active set.
pWrk	A symmetric work array. The pWrk argument must have the same data type as target. In C/C++, this contains max(nreduce/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements.
pSync	A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The nreduce argument determines the number of separate reduction 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 values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pWrk and pSync arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The target array on all PEs in the active set is ready to accept the results of the reduction.

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

- The target array is updated.
- The values in the pSync array are restored to the original values.

The function `shmem_double_sum_to_all` performs a reduction applaying the sum function to `doubles` values distributed across the PEs.

The function `shmem_float_sum_to_all` performs a reduction applaying the sum function to `float` values distributed across the PEs.

The function `shmem_int_sum_to_alql` performs a reduction applaying the sum function to `integer` values distributed across the PEs.

The function `shmem_long_sum_to_all` performs a reduction applaying the sum function to `long` values distributed across the PEs.

The function `shmem_longdouble_sum_to_all` performs a reduction applaying the sum function to `long double` values distributed across the PEs.

The function `shmem_longlong_sum_to_all` performs a reduction applaying the sum function to `long long` values distributed across the PEs.

The function `shmem_short_sum_to_all` performs a reduction applaying the sum function to `short` values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_int_xor_to_all, shmem_long_xor_to_all, shmem_longlong_xor_to_all, shmem_short_xor_to_all – perform a logical exclusive OR function across a set of processing elements (PEs)

SYNOPSIS

```
#include <shmem.h>
void shmem_int_xor_to_all(int *target, int *source, int nreduce,
                          int PE_start, int logPE_stride,
                          int PE_size, int *pWrk, long *pSync);
void shmem_long_xor_to_all(long *target, long *source,
                           int nreduce, int PE_start,
                           int logPE_stride, int PE_size,
                           long *pWrk, long *pSync);
void shmem_longlong_xor_to_all(long long *target,
                               long long *source, int nreduce,
                               int PE_start, int logPE_stride,
                               int PE_size, long long *pWrk,
                               long *pSync);
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);
```

PARAMETERS

`target`	A symmetric array, of length `nreduce`, to receive the result of the reduction operation. The data type of `target` should match that implied in the SYNOPSIS section.
`source`	A symmetric array, of length `nreduce`, that contains one element for each separete reduction operation. The `source` argument must have the same data type as `target`.
`nreduce`	The number of elements in the `target` and `source` array.
`PE_start`	The lowest virtual PE number of the active set of PEs.
`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE number in the active set.
`PE_size`	The number of PEs in the active set.
`pWrk`	A symmetric work array. The `pWrk` argument must have the same data type as `target`. In C/C++, this contains `max(nreduce/2+1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync	A symmetric work array. In C/C++, pSync must be of type long and size _SHMEM_REDUCE_SYNC_SIZE. Every element of this array must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory reduction routines compute one or more reductions across symmetric arrays on multiple virtual PEs. A reduction performs an associative binary operation across a set of values. The nreduce argument determines the number of separate reduction 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 values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the active set. Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pWrk and pSync arrays on all PEs in the active set are not still in use from a prior call to a collective shared memory routine.
- The target array on all PEs in the active set is ready to accept the results of the reduction.

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

- The target array is updated.
- The values in the pSync array are restored to the original values.

The function shmem_int_xor_to_all performs a reduction applaying the logical exclusive OR operator on integer values distributed across the PEs.

The function shmem_long_xor_to_all performs a reduction applaying the logical exclusive OR operator on long values distributed across the PEs.

The function shmem_longlong_xor_to_all performs a reduction applaying the logical exclusive OR operator on long long values distributed across the PEs.

The function shmem_short_xor_to_all performs a reduction applaying the logical exclusive OR operator on short values distributed across the PEs.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

2.11 Collective Communication

The collective communication functions are listed in Table 2.8.

Table 2.8: Collective Communication Functions

Name	Description
`shmem_broadcast`	Broadcasts a block of data having 64 bit storage class
`shmem_broadcast32`	Broadcasts a block of data having 32 bit storage class
`shmem_broadcast64`	Broadcasts a block of data having 64 bit storage class
`shmem_collect`	Concatenates blocks of data having 64 bit storage class
`shmem_collect32`	Concatenates blocks of data having 32 bit storage class
`shmem_collect64`	Concatenates blocks of data having 64 bit storage class
`shmem_fcollect`	Concatenates blocks of data having 64 bit storage class
`shmem_fcollect32`	Concatenates blocks of data having 32 bit storage class
`shmem_fcollect64`	Concatenates blocks of data having 64 bit storage class

Collective communication routines operate on the same data object on multiple PE. The Shmem supports two different type of collective communication as explained below:

- *Broadcast* routines (i.e. `shmem_broadcast`) that are used to broadcast a block of data from one processing element (named the *root* of the operation) to a set of PEs.
- *Concatenation* routines (i.e. `shmem_collect`) that are used to concatenate data items distributed over a set of PEs.

The collective communication functions are described in detail on the following pages.

NAME

shmem_broadcast, shmem_broadcast32, shmem_broadcast64 – broadcasts a block of data from one processing element (PE) to one or more target PEs

SYNOPSIS

```
#include <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);
void shmem_broadcast64(void *target, void *source, int nlong,
                       int PE_root, int PE_start,
                       int logPE_stride, int PE_size,
                       long *pSync);
```

PARAMETERS

`target`	A symmetric data object used to receive the data broadcasted by the processing element specified by `PE_start`. For `shmem_broadcast` and `shmem_broadcast64` the data type of `target` can be any type that has an element size of 64 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` and `shmem_broadcast64`, this is the number of 64-bit words. For `shmem_broadcast32` this is the number of 32-bit halfwords.
`PE_root`	Zero-based ordinal of the PE, with respect to the active set, from which the data is copied. Must be greater than or equal to `0` and less than `PE_size`.
`PE_start`	The lowest virtual PE number of the active set of PEs.
`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE numbers in the active set.
`PE_size`	The number of PEs in the active set.
`pSync`	A symmetric work array. In C/C++, `pSync` must be of type `long` and size `_SHMEM_REDUCE_SYNC_SIZE`. Every element of this array

must be initialized with the value _SHMEM_SYNC_VALUE before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

The shared memory 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 values of arguments PE_root, PE_start, logPE_stride, and PE_size must be equal on all PEs in the active set. The same target and source data objects and the same pSync work array must be passed to all PEs in the active set. Before any PE calls a broadcast routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this):

- The pSync arrays on all PEs in the active set is not still in use from a prior call to a broadcast routine.
- The target array on all PEs in the active set is ready to accept the broadcast data.

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

- If the current PE is not the root PE, the target data object is updated.
- The values in the pSync array are restored to the original values.

SEE ALSO

shmem_barrier(3) **shmem_barrier_all**(3)

NAME

shmem_collect, shmem_collect32, shmem_collect64, shmem_fcollect, shmem_fcollect32, shmem_fcollect64 – concatenates blocks of data from multiple processing elements (PEs) to an array in every PE

SYNOPSIS

```
#include <shmem.h>
void shmem_collect(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_collect64(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_fcollect32(void *target, void *source, int nlong,
                      int PE_start, int logPE_stride, int PE_size,
                      long *pSync);
void shmem_fcollect64(void *target, void *source, int nlong,
                      int PE_start, int logPE_stride, int PE_size,
                      long *pSync);
```

PARAMETERS

`target`	A symmetric array. The `target` argument must be large enough to accept the concatenation of the `source` arrays on all PEs. For `shmem_collect`, `shmem_collect64`, `shmem_fcollect` and `shmem_fcollect64`, the data type of `target` can be any type that has an element size of 64 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 the `source` array. The `nlong` argument must be equal on all PEs for `shmem_collect64`, `shmem_fcollect`, `shmem_fcollect32`, and `shmem_fcollect64`. 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.

`logPE_stride`	The log (base 2) of the stride between consecutive virtual PE numbers in the active set.
`PE_size`	The number of PEs in the active set.
`pSync`	A symmetric work array. In C/C++, `pSync` must be of type `long` and size `_SHMEM_REDUCE_SYNC_SIZE`. Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` before any of the PEs in the active set enter the reduction routine.

DESCRIPTION

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

SEE ALSO

`shmem_broadcast``(3)`

2.12 Address Manipulation

The address manipulation functions listed in Table 2.9 are not supported in this implementation.

Table 2.9: Address Manipulation Functions

Name	Description
`shmem_ptr`	Returns a pointer to a data object on a remote PE
`shmem_stack`	Makes a stack address remotely accessible

Entry points for these functions are provided in the library. The functions will generate an exception if called.

2.13 Control Data Cache

The functions for controls data cache cache are listed in Table 2.10.

Table 2.10: Control Data Cache Functions

Name	Description
`shmem_clear_cache_inv`	Disables automatic cache coherency mode
`shmem_set_cache_inv`	Enables automatic cache coherency mode
`shmem_set_cache_line_inv`	Enables automatic line cache coherency mode
`shmem_udcflush`	Makes the entire user data cache coherent.
`shmem_udcflush_line`	Makes coherent a cache line

These routines are supplied for compatibility with the Cray Shmem library. They perform no operations and returns to the caller successfully. The control data cache functions are described in detail on the following pages.

NAME

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

SYNOPSIS

```
#include <shmem.h>

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

DESCRIPTION

These routines are suplied for compatibility with the Cray Shmem library. They perform NULL operations and return to the caller successfully.

3

Programming Examples

3.1 Introduction

This chapter contains a programming example which makes use of the facilities of the Shmem library. This programming example implements a multiprocess version of the UNIX program `ping` using the Shmem routines.

`ping` sends packets across the network to elicit a response from a specified network host and prints out timing statistics for the round-trip (sending a packet and getting a response).

The example program extends `ping` to work on multiple network hosts by running processes in parallel on a number of processors. The processes form pairs and each process in the pair pings the other. After a user specified number of pings, one of the processes in each pair prints its timing statistics.

The following sections describe how the program is implemented. The complete program listing is given in Section 3.10.

3.2 The Command Line Interface

This is the command line interface for the program, `sping`.

```
sping -n number[k|K|m|M] -eh nwords [maxWords [incWords]]
```

The options for the programs are:

`-n` *`number`*`[k|K|m|M]`

Specifies the number of times to ping. The *`number`* may have a `k` or an `m` appended to it (or their upper case equivalents) to denote multiples of 1024 and 1,048,576 respectively. By default, the program pings 10,000 times.

`-e` Instructs every process to print their timing statistics.

`-h` Displays the list of options.

`nwords` [*`maxWords`* [*`incWords`*]]

`nwords` specifies to `sping` how many words there are in each packet. If *`maxWords`* is given, it specifies a maximum number of words to send in each packet and invokes the following behavior. After each *n* repetitions (as specified with the `-n` option), the packet size is increased by `incWords` (the default is a doubling in size) and another set of repetitions is performed until the packet size exceeds `maxWords`. This means that if neither of the optional parameters are specified, only one set of repetitions is performed.

3.3 Program Output

At the start of the program, if printing has been enabled for all processes with the `-e` option, a message like this is displayed by each process.

```
1(8): Shmem PING reps 250000 minWords 64 maxWords 128 incWords 32
```

where `1` is the process's identity number (i.e. the processing element or PE number) and `8` gives the number of processes running in parallel.

After each set of repetitions, timing statistics are displayed like this:

```
1 pinged 0:       64 words     10.14 uSec    50.49 MB/s
```

This indicates that process 1 pinged process 0 with 64 word packets. The pinging took 10.14 microseconds giving a rate of 50.49MBytes per second.

If printing has been enabled for all processes with the `-e` option, this message is displayed by each process. By default, only one process in each pair displays the message.

3.4 Header Files and Variables

The header files and variables used by the program are shown here. The variables are declared in `main`.

```
#include <stdio.h> [1]
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>

#include <shmem.h>

int main (int argc, char *argv[])
{
    double         t,tv[2]; [2]

    int            reps = 10000; [3]
```

```
    int             minWords = 1;
    int             maxWords = 1;
    int             incWords;

    int             proc; 4
    int             peer;
    int             nproc;

    long           *rbuf; 5
    long           *tbuf;

    int             doprint = 0; 6
    char           *progName;
    int             nwords, c, r, i;
    ...
}
```

The header files and variables are described here.

1 Besides the standard C header files, the `shmem.h` header file is required for the Shmem libraries.

2 The two time variables are used to time each set of repetitions of writing and reading a shared variable. The `tv` array is used to record two times using the function `gettimeofday`:

1. The time before the set of repetitions begins.
2. The time after the set of repetitions has ended.

The variable `t` is used to hold the difference between these two readings. All the time values are expressed in microseconds.

3 This group of variables is used to control how many times the process pings its opposite number and the size of packets sent. The variable `reps` is set to the number of repetitions requested with the `-n` option. It has a default setting of 10,000.

The next three variables hold the minimum, maximum and increment values for the packet size. They are used when more than one set of repetitions is requested. The variable `incWords` is used to iterate from `minWords` to `maxWords` during a set of repetitions.

4 These variables are used to identify by means of their PE number the process and its peer or opposite number to which it write a shared variable, and to hold the total number of processing elements.

5 The variable `rbuf` is a pointer to a shared buffer. A processing element uses this buffer pointer to write data in the memory of its peer. The variable `tbuf` is a pointer to a buffer containing the data used to fill the shared buffer `rbuf`.

6 The variable `doprint` is used to enable (`1`) or disable (`0`) the printing of results by all the processes.

The `progName` variable is used to extract the name of the program for use with the standard UNIX style `-h` option and `Usage` message which is displayed when the program is called with the wrong arguments.

The remaining four variables are general purpose iteration variables.

3.5 Argument Checking

The first section of `main` is concerned with checking the arguments passed to the program on the command line.

```
int main(int argc, char *argv[]) {
    ...
    for (progName = argv[0] + strlen (argv[0]); 1
         progName > argv[0] && *(progName - 1) != '/';
         progName--)
         ;

    while ((c = getopt (argc, argv, "n:eh")) != -1) 2
        switch (c) {
        case 'n':
            if ((reps = getSize (optarg)) <= 0)
                usage (progName);
            break;

        case 'e':
            doprint++;
            break;

        case 'h':
            help (progName);

        default:
            usage (progName);
        }

    if (optind == argc) 3
        minWords = 1;
    else if ((minWords = getSize (argv[optind++])) <= 0)
        usage (progName);

    if (optind == argc)
        maxWords = minWords;
    else if ((maxWords = getSize (argv[optind++])) < minWords)
        usage (progName);

    if (optind == argc)
        incWords = 0;
    else if ((incWords = getSize (argv[optind++])) < 0)
        usage (progName);
    ...
}
```

1 The program name is passed in as `argv[0]`, the first string on the command line. This string may take the form of a pathname, such as `/opt/rms/example/sping`. The `progname` variable is set to point to the end of the program name. The loop then steps the variable backwards, one character at a time, until either a filename separator (`/`) or the beginning of the name is reached. This leaves `progname` pointing at the start of the program name.

2 The `while` loop steps through the options given on the command line.

- If the `-n` option has been used, the variable `reps` is set to the requested number of repetitions after a check that the number is greater than `0`. If the number is invalid, the `usage` function is called. This merely displays the command line syntax for the program and then exits.
- If the `-e` option has been used, the variable `doprint` is incremented. This variable is used later to enable or disable the printing of statistics.
- The `-h` option calls the `help` function which displays the command line syntax for the program and explains the meaning of the various options (or flags), like this.

```
Usage: sping [flags] nwords [maxWords] [incWords]

   Flags may be any of
      -n number              repetitions to time
      -e                     everyone print timing info
      -h                     print this info

   Numbers may be postfixed with 'k' or 'm'
```

- If any other options beside the three mentioned here are given, the function `usage` is called to display the correct command line syntax and then exit.

3 The three `if` statements determine whether the optional arguments for specifying a varying packet size have been set. The variable `optind` is defined externally and included by the header files at the start of the program. After stepping through all the options with the `while` loop, `optind` indexes the first argument in `argv`.

The first argument should be `nwords`, the number of words in each packet. If the user has not specified this argument, the program continues rather than exiting but assumes a value of `1`. Note that the value is assigned to `minWords` rather than to the variable `nwords`. Later on, the value is transferred to `nwords` when it acts as an iteration variable.

3.6 Initialization

The next section of `main` is concerned with initializing the process to use the Shmem library and setting up a target shared variable.

```
int main (int argc, char *argv[]) {
    ...
    if (!(rbuf = (long *)malloc(maxWords * sizeof(long)))) 1
      {
        perror ("Failed memory allocation");
        exit (1);
      }

    if (!(tbuf = (long *)malloc(maxWords * sizeof(long))))
     {
        perror ("Failed memory allocation");
        exit (1);
     }

    for (i = 0; i < maxWords; i++)
       tbuf[i] = 1000 + (i & 255);

    memset (rbuf, 0, maxWords * sizeof (long));

    shmem_init(); 2

    proc = my_pe(); 3
    nproc = num_pes();

    if (nproc == 1)
      exit (0);
    ...
}
```

The initialization process is as follows.

1 The process allocates memory for the two message buffers `rbuf` and `tbuf` using the `malloc` function. The buffers are used as the destination and source in the Shmem remote write operation. Pointers to them are passed to the Shmem library functions.

If a maximum number of words for the packet size is specified on the command line to `sping`, the process allocates a buffer of this size. By default, the buffers are 8 bytes (1 word). The transmit buffer `tbuf` is initialized by writing a sequence of numbers to it. The remote buffer `rbuf` is initialized to zero.

2 The process calls `shmem_init` to initialize itself to use the Shmem library.

3 The process calls the functions `my_pe` and `num_pes` to determinate its PE number and to find out how many processes are running in parallel. If it is the only process, it exits as there is no-one for it to ping.

3.7 Establishing the Peer Group

Before starting the first (and possibly only) set of repetitions, the processes must synchronize and group themselves into pairs.

```
int main(int argc, char *argv[]) {
    ...
    if (doprint) 1
        printf ("%d(%d): Shmem PING reps %d
                minWords %d maxWords %d incWords %d\n",
                proc, nproc, reps, minWords, maxWords, incWords);

    shmem_barrier_all(); 2

    peer = proc ^ 1; 3
    if (peer >= nproc)
        doprint = 0;
    ...
}
```

1 If all the processes have been enabled for printing with the `-e` option, each prints a message to confirm its identity, the number of processes in the program and the program parameters.

2 Before starting to ping each other, the processes synchronize, that is to say, each waits in the call to `shmem_barrier_all` until all have made the call. This guarantees that all the processes are initialized and ready to write and read shared variables before any one of them starts to ping another.

3 In order to ping each other, the processes split up into pairs. Each process determines its opposite number or `peer` simply by an exclusive-OR of its own PE number identifier with the constant `1`. The processes have PE identifier numbered from `0` to `nproc-1`, where `nproc` is the number of processes in the program. With an uneven number of processes, one will have no peer. This can be determined by checking that the peer's PE number is in the range of valid PEs identifiers. This singleton is disabled from printing.

3.8 Writing Shared Variables

In the final section of `main`, the process pings its peer a given number of times using the Shmem functions.

```
int main(int argc, char *argv[]) {
   ...
    for (nwords = minWords;
         nwords <= maxWords;
         nwords = incWords ? nwords + incWords : nwords ? 2 * nwords : 1)  { 1
       r = reps;

       shmem_barrier_all(); 2
```

```
        tv[0] = gettime();

        if (peer < nproc) { 3
           if (proc & 1)
            {
              r--;
              shmem_wait(&rbuf[nwords-1], 0);
              rbuf[nwords-1] = 0;
            }

           while (r-- > 0) 4
            {
              shmem_put(rbuf, tbuf, nwords, peer);
              shmem_wait(&rbuf[nwords-1], 0);
              rbuf[nwords-1] = 0;
            }

           if (proc & 1)
            shmem_put(rbuf, tbuf, nwords, peer);
         }

         tv[1] = gettime(); 5

         t = dt (&tv[1], &tv[0]) / (2 * reps);

         shmem_barrier_all();

         printStats (proc, peer, doprint, nwords, t);
    }

    shmem_barrier_all(); 6
    exit (0);
}
```

The Shmem library functions to access a shared variable are described here.

1 The `for` loop controls how many sets of repetitions are performed. In each set of repetitions, a message containing `nwords` words is written from one process to its peer for the number of times specified by `reps`.

The first time through the loop, `nwords` is set to `minWords`. This was initialized earlier (see Section 3.5) to the value the user entered for `nwords` on the command line (by default, `1`).

On subsequent iterations, the value of `nwords` is incremented by the value of `incWords`. If no value was specified for `incWords` on the command line, the original value of `nword` is doubled or, if `nwords` was unspecified, it is set to `1`.

If the user specified `maxWords`, the `for` loop is iterated until `nwords` exceeds the value of `maxWords`. If not, the loop is only executed once.

2 Before the processes begin to time how long the ping operation takes, they synchronize using `shmem_barrier_all`. This ensures

that they are all ready to start sending and receiving messages at the same time.

The timing is done by calling twice the function `gettime`: one before the remote writes start and one when they have finished.

3 After testing that the process has a peer (this test has to be repeated in here since all the processes must participate in the synchronization), the read/write operations on shared variables can begin. The odd numbered processes (`proc & 1`) start first by waiting that the shared variable is modified by the peer.

The call to `shmem_wait` blocks the process until the value stored in the `nwords-1` postion of the buffer `rbuf` is modified by the peer. The call to `shmem_wait` specifies:

- The address of a remotely accessible variable that is being updated by a remote processing element.
- The value `V` to be compared with the value `S` stored in the remotely accessible variable. The process blocks until `S` and `V` remain equal, that is until a remote processing element write a different value in the shared variable.

The buffer `rbuf` was initialised erlier to `0` (see Section 3.6) and thus the process blocks until the peer writes the shared buffer with a value different from `0`. When the process returns from the function `shmem_wait` it sets the (`nwords-1`)-th position of the shared buffer to `0` preparing the next iteration.

4 In the `while` loop both the odd and even numbered processes write the shared variable and then wait that the peer executed the remote write. The number of repetitions `r` is decremented each time. The call to `shmem_put` specifies:

- The address of the remote variable to the be updated on the remote PE (i.e. `rbuf`)
- The address of the local variable containing the data to be copied on the remote variable (i.e. `tbuf`)
- The number of elements in the local and remote variables (i.e. `nwords`)
- The PE number of the remote processing element where the local variable will be copied.

When the process has written the remote variable `rbuf` it waits until the peer performs a write on the its local buffer. This is done calling the `shmem_wait` function. Once that the process returns from this function it sets the (`nwords-1`)-th element of the shared buffer to `0` preparing for the next iteration.

Finally, the odd numbered process performs the final write on the remote buffer. By making the odd numbered processes wait for a

remote write to begin with while the even numbered processes write the remote target, deadlock is avoided.

5 After the set of repetitions, the process calls the function `gettime` again. It calculates the time taken for one ping in each direction (the difference between the two timer readings divided by the number of repetitions). This value, expressed in microseconds, is halved to get the value for a ping in one direction.

Before the processes print the results, they synchronise again. This means that all the results are displayed at roughly the same time and the printing does not interfere with the network performance.

6 When the process has come out of the `for` loop, it synchronises with its peers again before exiting.

3.9 Subsidiary Functions

The subsidiary functions make no use of the Shmem library.

`getSize` This function checks whether the user has suffixed the number of repetitions, specified on the command line with the `-n` option, with either a `k` or `K` (for kilobytes) or `m` or `M` (for megabytes). If it finds a suffix, it multiplies the number as appropriate (a left shift by one place multiplies by 2).

`dt` This function returns the difference between its two arguments.

`usage` This function prints out the command line syntax for the program and then exits.

`help` This function prints out the command line syntax for the program and enumerates the various options before exiting.

`printStats` This functions displays the timing statistics generated during each set of repetitions. Unless printing is enabled for all processes with the `-e` option, only the odd numbered processes have their statistics displayed.

3.10 Program Listing

This section shows the program in its entirety.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>

#include <shmem.h>
```

```
int getSize (char *str)
{
   int             size;
   char            mod[32];

   switch (sscanf (str, "%d%1[mMkK]", &size, mod))
   {
   case 1:
      return (size);

   case 2:
      switch (*mod)
      {
      case 'm':
      case 'M':
         return (size << 20);

      case 'k':
      case 'K':
         return (size << 10);

      default:
         return (size);
      }

   default:
      return (-1);
   }
}

double gettime()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return (tv.tv_sec * 1000000 + tv.tv_usec);
}

double dt (double *tv1, double *tv2)
{
   return (*tv1 - *tv2);
}

void usage (char *name)
{
   fprintf (stderr, "Usage: %s [flags] nwords [maxWords] [incWords]\n", name);
   fprintf (stderr, "       %s -h\n", name);
   exit (1);
}

void help (char *name)
{
   printf ("Usage: %s [flags] nwords [maxWords] [incWords]\n", name);
   printf ("\n");
   printf ("   Flags may be any of\n");
   printf ("      -n number              repititions to time\n");
   printf ("      -e                     everyone print timing info\n");
   printf ("      -h                     print this info\n");
   printf ("\n");
   printf ("   Numbers may be postfixed with 'k' or 'm'\n");
```

```
   printf ("\n");
   exit (0);
}

void printStats (int proc, int peer, int doprint, int now, double t)
{
   if (doprint || (proc & 1))
       printf("%3d pinged %3d: %8d words %9.2f uSec %8.2f MB/s\n",
              proc, peer, now, t, sizeof(long)*now/(t));
}

int main (int argc, char *argv[])
{
    double         t,tv[2];

    int            reps = 10000;
    int            doprint = 0;
    char          *progName;
    int            minWords = 1;
    int            maxWords = 1;
    int            incWords;
    int            nwords;
    int            nproc;
    int            proc;
    int            peer;
    int            c;
    int            r;
    int            i;
    long          *rbuf;
    long          *tbuf;


    for (progName = argv[0] + strlen(argv[0]);
         progName > argv[0] && *(progName - 1) != '/';
         progName--)
        ;

    while ((c = getopt (argc, argv, "n:eh")) != -1)
        switch (c)
        {
        case 'n':
            if ((reps = getSize (optarg)) <= 0)
                usage (progName);
            break;

        case 'e':
            doprint++;
            break;

        case 'h':
            help (progName);

        default:
            usage (progName);
        }

    if (optind == argc)
        minWords = 1;
    else if ((minWords = getSize (argv[optind++])) <= 0)
        usage (progName);
```

```
if (optind == argc)
    maxWords = minWords;
else if ((maxWords = getSize (argv[optind++])) < minWords)
    usage (progName);

if (optind == argc)
    incWords = 0;
else if ((incWords = getSize (argv[optind++])) < 0)
    usage (progName);

if (!(rbuf = (long *)malloc(maxWords * sizeof(long))))
  {
    perror ("Failed memory allocation");
    exit (1);
  }
memset (rbuf, 0, maxWords * sizeof (long));

  if (!(tbuf = (long *)malloc(maxWords * sizeof(long))))
    {
      perror ("Failed memory allocation");
      exit (1);
    }

shmem_init();
proc = my_pe();
nproc = num_pes();

if (nproc == 1)
    exit (0);


for (i = 0; i < maxWords; i++)
    tbuf[i] = 1000 + (i & 255);

if (doprint)
    printf ("%d(%d): Shmem PING reps
             %d minWords %d maxWords %d incWords %d\n",
            proc, nproc, reps, minWords, maxWords, incWords);

shmem_barrier_all();

peer = proc ^ 1;
if (peer >= nproc)
    doprint = 0;

for (nwords = minWords;
     nwords <= maxWords;
     nwords = incWords ? nwords + incWords : nwords ? 2 * nwords : 1)
{
    r = reps;

    shmem_barrier_all();

    tv[0] = gettime();

    if (peer < nproc)
    {
        if (proc & 1)
```

```
                {
                  r--;
                  shmem_wait(&rbuf[nwords-1], 0);
                  rbuf[nwords-1] = 0;
                }

                while (r-- > 0)
                  {
                  shmem_put(rbuf, tbuf, nwords, peer);
                  shmem_wait(&rbuf[nwords-1], 0);
                  rbuf[nwords-1] = 0;
                }

                if (proc & 1)
                  shmem_put(rbuf, tbuf, nwords, peer);
            }

            tv[1] = gettime();


            t = dt (&tv[1], &tv[0]) / (2 * reps);

            shmem_barrier_all();

            printStats (proc, peer, doprint, nwords, t);
        }

        shmem_barrier_all();
        exit (0);
    }
```

Glossary

Abbreviations

API Application Program Interface — specification of interface to software package (library).

CFS Cluster File System — the file system for Tru64 UNIX clusters.

CGI Common Gateway Interface — a standard method for generating HTML pages dynamically from an application so that a Web server and a Web browser can exchange information. A CGI script can be written in any language and can access various types of data, for example, a SQL database.

CPU Central Processing Unit — the part of the computer that executes the machine instructions that make up the various user and system programs.

CRC Cyclic Redundancy Check — a method of error detection.

CVS Concurrent Versions System — a revision control utility for managing software releases and controlling the concurrent editing of files by multiple software developers.

DIMM Dual In-Line Memory Module.

DMA Direct Memory Access — high performance I/O technique where peripherals read/write memory directly and not through a CPU.

GNU GNU's Not UNIX — A UNIX-like development effort of the Free Software Foundation, headed by Richard Stallman.

HTML HyperText Markup Language — a generic markup language, comprising a set of tags, that enables structured documents to be delivered over the World Wide Web and viewed by a browser.

HTTP	HyperText Transfer Protocol — a communications protocol commonly used between a Web server and a Web browser together with a URL (Uniform Resource Locator).
LED	Light-Emitting Diode.
MIMD	Multiple Instruction, Multiple Data — parallel processing computer architecture characterised as having multiple processors each (potentially) executing a different instruction sequence on different data.
MMU	Memory Management Unit — part of CPU that provides protection between user processes and support for virtual memory.
MPI	Message Passing Interface — parallel processing API.
MPP	Massively Parallel Processing — processing that involves the use of a large number of processors in a coordinated fashion.
PCI	Peripheral Component Interconnect — the Elan is connected to a node through this interface.
PDF	Portable Document Format — the page description language used by Adobe Acrobat, derived from PostScript, for displaying pages on the screen.
PTE	Page Table Entry — an entry in the page table which maps the base address of a page to physical memory.
RISC	Reduced Instruction Set Computer — a computer whose machine instructions represent relatively simple operations that can be executed very quickly.
RMS	Resource Management System — Quadrics software for managing clusters of UNIX nodes.
SDRAM	Synchronous Dynamic Random Access Memory — high performance computer memory architecture.
Shmem	A one-sided (put/get) inter-process communication interface used on high-performance parallel systems.
SMP	Symmetric MultiProcessor — a computer whose main memory is shared by more than one processor.
SNMP	Simple Network Management Protocol — a protocol used to monitor and control devices on the Internet.
SQL	Structured Query Language — a database language.

TLB Translation Lookaside Buffer — part of the MMU that caches the result of virtual to physical address translations to minimise translation times in subsequent accesses to the same page.

URL Uniform Resource Locator — a standard protocol for addressing information on the World Wide Web.

UTC Coordinated Universal Time[1] — on UNIX systems it is represented as the time elapsed in seconds since January 1st, 1970 at 00:00:00.

Terms

barrier A synchronisation point in a parallel computation that all of the processes must reach before they are allowed to continue.

bisectional bandwidth

The worst case bandwidth across the diameter of the network.

block A thread that blocks without relinquishing the processor until a specified event occurs.

critical section A section of program statements that can yield incorrect results if more than one thread tries to execute the section at the same time.

Elan memory The SDRAM on the Elan card.

event A parallel-processing synchronisation primitive implemented by the Elan card.

Flit A communications cycle unit of information.

HTTP cookies Cookies provide a general mechanism that HTTP server-side connections use to store and to retrieve information on the client side of the connection.

main memory The memory normally associated with the main processor, that is to say, memory on the CPU's high speed memory bus.

main processor The main CPU (or CPUs for a multi-processor) of a node, typically an Alpha™ 21264.

management network

A private network used by the RMS daemons for control and diagnostics.

[1] Used to be called GMT.

multirail system

A system that has more than one Elan card connected to each node, each Elan card being connected to a different switch network.

multi-threaded program

A multi-threaded program is one that is constructed such that, during its execution, multiple sequences of instructions are executed concurrently (possibly by different CPUs). Each thread of execution has a separate stack but otherwise they all share the same address space.

node A system with memory, one or more CPUs and one or more Elan cards running an instance of the operating system.

poll Loop and check on each loop whether a specified event has occurred.

rank An integer value that identifies a single process from a set of parallel processes.

reduce Combine the results of a parallel computation into a single value.

remote memory The memory (Elan card or main) of a node when accessed by another node over the network.

resource A set of CPUs allocated to a user to run one or more parallel jobs.

slice A local copy of a global object.

switch network The network constructed from the Elan cards and Elite cards.

thread An independent sequence of execution. Every host process has at least one thread.

virtual memory A feature provided by the operating system, in conjunction with the MMU, that provides each process with a private address space that may be larger than the amount of physical memory accessible to the CPU.

virtual process A (possibly multi-threaded) component of a parallel program executing on a node.

word A 32-bit value.

Index