OpenSHMEM Reference Library Implementation

Tony Curtis*

Computer Science Department, University of Houston

*Email address: arcurtis@mail.uh.edu

March 31, 2011

Contents

1 Sponsorship

Work on the OpenSHMEM project is sponsored by the Oak Ridge National Laboratory Extreme Scale System Center and the U.S. Department of Defense.

2 Introduction

This document is an overview of the implementation strategy for the initial, reference version of what will become OpenSMHEM. We will discuss the concept of a partitioned global address space for completeness.

3 Terminology

A SHMEM program consists of a number of processors executing separate processes. A processor is referred to as a "processing element", or PE. All PEs run the same program in the SPMD model, although they can discover which position, or rank, they occupy within the program and diverge in behavior.

The number of PEs participating in the program is set at launch-time (although not all PEs need to do work). PEs are numbered monotonically increasing from 0 up to $N-1$, where N is the total number of PEs. PEs are assumed to be equidistant from each other for communication purposes, no topological information is currently exposed.

Communication occurs through point-to-point one-sided routines and collective operations. A one-sided operation is a communication in which one PE transfers data to another PE, but the "other" PE does not participate: the data transfer does not cause the other PE to be interrupted to acknowledge the transfer.

4 Partitioned Global Address Space

Parallel programs running in a distributed environment access both local and remote data. The model used to construct the program can either expose or hide this distribution. Exposed models include that of MPI and PVM, in which explicit messages are required to pass data between processors participating in a parallel program. Hidden models include those with a Global Address Space (GAS), in which there appears to be memory accessible from all processors. This memory may be physically accessible, or may in fact be made available through I/O operations over network interconnects.

SHMEM provides a symmetric view of memory, in which processors allocate variables in concert but have their own copies. A processor can then "put" or "get" data to or from other processors by requesting a specific variable on another processor. SHMEM provides for address translation when required to allow a variable allocated by one processor to be accessed by another, because in a number of environments it is not guaranteed that address spaces are uniform.

This allocation-in-concert of separate variables is termed Partitioned Global Address Space (PGAS).

Clusers that use interconnects with remote direct memory access (rDMA) are of particular interest to the PGAS community as they provide hardware off-load capability to avoid interrupts during one-sided communications.

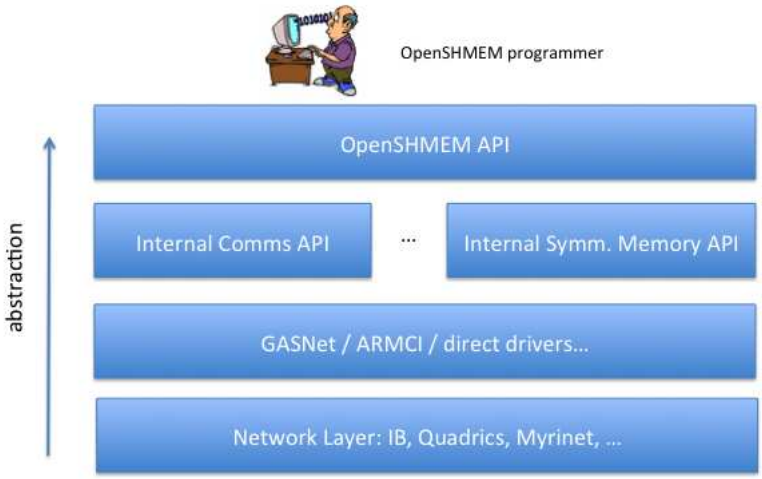
5 SHMEM History and OpenSHMEM

The SHMEM communications library was originally developed as a proprietary application interface by Cray for their T3D systems and subsequently the T3E models. These systems typically consisted of a memory subsystem with a logically shared address space over physically distributed memories, a memory interconnect network, a set of processing elements (PEs), a set of input-output gateways, and a host subsystem. The systems were designed to support latency hiding mechanisms such as prefetch queues, remote stores and the Block Transfer Engine (BLT). The prefetch queues allowed the users to issue multiple asynchronous single-word reads which could overlap with computation. Remote stores enabled PEs to directly write to other PE's memory asynchronously, while the BLT could hide latency while transferring blocks of data efficiently between local memory and remote memory locations. The explicit shared memory programming method allowed structured communication via shared memory on Cray MPP systems.

SHMEM was later adapted by SGI for its products based on the Numa-Link architecture and included in the Message Passing Toolkit (MPT). Other SHMEM implementations grew out of the SGI and Cray implementations, including Quadrics, HP, IBM, gpSHMEM and SiCortex, but diverged from the original libraries as they developed. These implementations of the SHMEM API support C, C++, and Fortran programs; however, the differences between SHMEM implementations' semantics and APIs are subtle, resulting in portability and correctness issues. U.S. Department of Defense funded a collaboration between Oak Ridge National Laboratory and the University of Houston to develop a specification for a uniform SHMEM API. The OpenSHMEM specification was announced to address the divergence of the SHMEM APIs.

6 The Reference OpenSHMEM Library

The overall structure of the reference library is shown below. We use an intermediate library such as GASNet or ARMCI to abstract away from a particular interconnect/platform, although there is nothing to stop a more direct approach being used. Internally, the reference implementation of OpenSHMEM provides private APIs for memory management, the communications layer, tracing/debugging and (eventually) support for adaptivity to choose things like barrier algorithm at run-time.



7 Implementation Strategy

In this section we talk about how the reference library was written and why certain implementation strategies were chosen.

7.1 GASNet

GASNet provides access to the symmetric memory areas. A memory management library marshals accesses to these areas during allocation and freeing of symmetric variables in user code, usually through a call like `shmalloc()`.

When you `gasnet_attach()` and ask for segment information, each PE has access to an array of segments, 1 segment per PE. Each PE initializes a memory pool within its own segment. The set up is handled either by GASNet internally ("fast"/"large" model) or by OpenSHMEM itself ("everything" model). The table of segments allows any PE to know the virtual location and size of the segment belonging to any other PE.

If the platform allows it, GASNet can align all the segments at the same address, which means that all PEs see the same address for symmetric variables and there's no address translation.

In the general case though, segments are not aligned (e.g. due to a security measure like process address space randomization by the OS). However, each PE can see the addresses of the segments of the other PEs locally, and can therefore do address translation.

Currently alignment is not checked for, so we're coding to the "worst case scenario". That just adds a *small* overhead if the segments are in fact aligned. The library should at some point introduce code that differentiates between aligned and non-aligned environments with optimized code for the former case (GASNet provides a macro you can test against).

7.1.1 Segment Models

The library currently has best support for the "everything" model. This model allows the entire process space to be addressed remotely. Communication with dynamically allocated data and with global data is equally easy.

For the "fast" and "large" models, only a specified area of the process memory is exposed for remote access. This means extra support has to be added to handle communication with global variables, because only the symmetric heap is exposed by GASNet.

For the SMP conduit, PSHM support is required to run parallel threaded programs with OpenSHMEM. This excludes the "everything" model.

7.2 Initialization

In `src/updown.c` we handle setting up the OpenSHMEM runtime, and eventual shutdown. Shutdown is implicit in user code, there is no call to do this in SGI SHMEM, so we register a handler to be called when `main()` exits. (Cray SHMEM has an explicit finalize call, however.) The segment exchange is a target for future optimization: in large programs, the start-up time will become burdensome due to the large number of address/size communication. Strategies for avoiding this include lazy initialization and hierarchical lookups.

7.3 Incorporating SHMEM into Programs

For C, the appropriate header file must be included: both the SGI-compliant `<mpp/shmem.h>` and the Quadrics-like `<shmem.h>` are handled for portability. We provide both the SGI `start_pes(int npes)` and `shmem_init(void)` interfaces, again for portability. The calls are synonyms. `start_pes()` just ignores its argument (consistent with SGI behavior. SGI indicates "npes" *should* be set to zero but we don't enforce that, merely note it). The number of PEs is taken from the invoking environment. Currently this number is assumed to be fixed throughout the lifetime of the program, but fault tolerance extensions could generalize this notion. Below are simple C and Fortran program templates:

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

int
main(int argc, char *argv[])
{
  int me, npes;

  start_pes(0);
  me = _my_pe();        /* which PE I am */
  npes = _num_pes();    /* how many PEs in program */
  printf("Hello from PE %d of %d\n", me, npes);
  return 0;
```

```
}
```

```
program hello
  include 'mpp/shmem.fh'
  integer me, npes

  call start_pes(0)
  me = my_pe()
  npes = num_pes()
  print *, 'Hello from PE ', me, ' of ', npes
end program hello
```

7.4 Communications Substrate

The OpenSHMEM library has been written to sit on top of any communications library that can provide the required functionalty. Initially we have targetted GASNet, and an ARMCI version is also planned. The `directory trunk/src/comms` provides implementations of the internal API. All subsequent references to GASNet should be read with an eye on the abstraction process.

7.5 Servicing Communications

A separate service thread handles network traffic. Waits and spins on variables yield to the service thread which is just a continuous poll on the communication layer. The initialization phase creates the service thread and the registered shutdown handler turns off the thread. If the "mainline" sleeps or otherwise can't handle communication, this thread takes up the work. It's meant to look like the way dedicated hardware solutions do this kind of thing. A routine provides the ability to send requests to the service thread. This is how `shmem_fence()` and `shmem_quiet()` are handled: we tell the service thread to wait for pending operations, and then resume polling.

7.6 Memory Management

Initially we tried to use the TLSF library:
`http://rtportal.upv.es/rtmalloc/`
but this proved to have weird interactions with Open-MPI. Tracking program progress with valgrind suggested that system memory calls were being intercepted.

So, following the Chapel lead,
`http://chapel.cray.com/`
we now use the "dlmalloc" library
`http://g.oswego.edu/dl/html/malloc.html`
to manage allocations in the symmetric memory space.

7.7 Point-to-point routines

Point-to-point operations are a thin layer on top of GASNet. The non-blocking put operations with implicit handles provide a way to subsequently fence and barrier. However, tracking individual handles explicitly with a hash table keyed on the address of symmetric variables may give better performance.

The Quadrics extensions that add non-blocking calls into the API proper have already been requested for the OpenSHMEM development.

7.8 Atomic Operations

Atomic operations include swaps, fetch-and-add and locks (discussed separately in 7.9). The first two are handled via GASNet's Active Messages. Increment was originally layered on top of add (increment = add 1, after all) but was rewritten with its own handlers. The payload for increment can be ever so slightly smaller than for add since there's no need to pass the value to add. In large applications, even such a small saving could add up (if you'll pardon the pun).

Eaarlier versions of the implementation had a single handler lock variable per operation (one for all adds, one for all increments, *etc.*). However, we've now added a hash table to dynamically allocate and manage per-target-address handler locks. Large-scale atomic operations, like add-scatters across multiple variables could easily benefit from this, as the lock granularity then permits concurrent discrete memory accesses.

7.9 Locks

OpenSHMEM provides routines to claim, release and test global locks. These can be used for mutual-exclusion regions. Our implementation is from the Quadrics library, which is a version of the Mellor-Crummy-Scott algorithm ("Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors" by John M. Mellor-Crummey and Michael L Scott). The locks are layered on top of OpenSHMEM primitives, so there are no Elan dependencies.

7.10 Barrier and broadcast

The initial version is naive, making the root of the broadcast a bottleneck. This is partly intentional, to allow Swaroop to explore better algorithms and work out how to demonstrate and document the improvements. We would like to collect some locality information inside the library to help decide communication order inside these algorithms: PEs that differ in rank by large amounts are likely to be further away topologically too, so by sending to more distant PEs first, we can stagger the network traffic and balance the latencies better. A proper measurement of "distance" is needed here. "hwloc" provides a per-system distance metric in NUMA terms. A simple extension could *e.g.* just multiply the distance by some constant when moving off-node to penalize network traffic.

7.11 Collects

The files `src/fcollect.c` and `src/collect.c` implement the collector routines (concatenating arrays on a set of PEs into a target array on all of those PEs).

fcollect is pretty easy since all the PEs must contribute the same amount of data. This means we can just pre-compute where each PE writes to their targets.

collect is harder because each PE can write different amounts. Thought of 2 ways of handling this:

1. initial exchange of sizes "from the left" so each PE can compute its write locations; then same as fcollect
2. wavefront: PEs wait for notification from PEs before them in the set (lower numbered). This passes the offsets across the set.

I used #2. #1 potentially generates a network storm as all PEs wait to work out where to write, then all write at once. #2 staggers the offset notification with a wave of writes moving up the PE numbers.

7.12 Reductions

Reductions coalesce data from a number of PEs into either a single variable or array on all participating PEs. The coalescing involves some kind of arithmetic or logic operation (e.g. sum, product, exclusive-or). Currently probably naive, using gets. A version with puts that can overlap communication and the computation of the reduction operation should be more scalable. However, the code is rather compact and all ops use the same template. A future version of OpenSHMEM may add user-defined reductions.

7.13 Address and PE Accessibility

OpenSHMEM allows us to test whether PEs are currently reachable, and whether addresses on remote PEs are addressable. GASNet is used to "ping" the remote PE and then we wait for an "ack" with a configurable timeout. Remains to be seen how useful this is, and whether it can be used for future fault tolerance issues.

7.14 Tracing Facility

This library contains "trace points" with categorized messages. These are listed in section 9

A high-resolution clock is maintained to timestamp such messages. Numerically sorting the output on the first field can thus help understand the order in which events happened.

7.15 C++

The C++ interface is basically the C one. There is one point of contention, namely complex numbers. The SGI documentation refers only to the use of C99 "complex" modifiers, not to C++'s `complex<T>`. The use of complex number routines (*e.g.* reductions) in C++ is thus not clearly specified.

7.16 Fortran

The Fortran interface is very similar to that of C. The names of various routines are different to accommodate the various type differences, e.g. `shmem_integer_put()` instead of `shmem_int_put()`.

The biggest difference is in the symmetric memory management routines. These have completely different names and parameters compared to the C interface.

The OpenSHMEM implementation handles Fortran with a very thin wrapper on top of C. Mostly this involves catching Fortran's pass-by-reference variables and dereferencing them in the underlying C call.

The main development has been on a CentOS platform with GNU 4.1.2-redhat. There seem to be some issues with this compilers' handling of cray-pointers: even the simplest programs (no OpenSHMEM content at all) produce a segmentation fault. Later versions (*e.g.* 4.5.0 ++) behave better.

8 Undefined Behavior

Many routines are currently specified only in terms of "correct" behavior. What happens when something goes wrong is not always specified. This section attempts to set out a few of these scenarios

- put to PE out of range: suppose we do a put to "right neighbor" (pe + 1). The highest-numbered PE will attempt to communicate with a PE that does not exist.
- library not initialized: virtually all OpenSHMEM routines will have major problems if the library has not been initialized. Implementations can handle this situation in different ways.

9 Environment Variables

The behavior of the OpenSHMEM library can be controlled via a number of environment variables. For SGI compatibility reasons, we support the "SMA" variables and our own new ones:

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

SHMEM_LOG_LEVELS: a comma, space, or semi-colon separated list of logging/-trace facilities to enable debugging messages. The facilities currently include the case-insensitive names:

Facility	Meaning
FATAL	something unrecoverable happened, abort
DEBUG	used for debugging purposes
INFO	something interesting happened
NOTICE	important event, but non-fatal (see below)
AUTH	when something is attempted but not allowed
INIT	set-up and tear-down of the program
MEMORY	symmetric memory information
CACHE	cache flushing operations
BARRIER	about barrier operations
BROADCAST	about broadcast operation
COLLECT	about collect and fcollect operation
REDUCE	about reduction operations
SYMBOLS	to inspect the symbol table information
LOCK	related to setting, testing and clearing locks
SERVICE	related to the network service thread
FENCE	tracing network fence events
QUIET	tracing network quiet events

SHMEM_LOG_FILE: a filename to which to write log messages. All PEs append to this file. The default is for all PEs to write to standard error. Per-PE log files might be an interesting addition.

SHMEM_SYMMETRIC_HEAP_SIZE: the number of bytes to allocate for the symmetric heap area. Can scale units with "K", "M" etc. modifiers. The default is 1M.

SHMEM_BARRIER_ALGORITHM: the version of the barrier to use. The default is "naive". Designed to allow people to plug other variants in easily and test.

SHMEM_BARRIER_ALL_ALGORITHM: as for SHMEM_BARRIER_ALGORITHM, but separating these two allows us to optimize if e.g. hardware has special support for global barriers.

`SHMEM_PE_ACCESSIBLE_TIMEOUT`: the number of seconds to wait for PEs to reply to accessiblity checks. The default is 1.0 (i.e may be fractional).

10 Compiling and Running Programs

The SGI SHMEM is provided as part of the Message-Passing Toolkit (MPT) in the ProPack suite. Compilation uses a standard C, C++ or Fortran compiler (*e.g.* GNU, Intel) and links against the SMA and MPI libraries.

In order to abstract the compilation and launching process for OpenSHMEM we have provided 4 wrapper programs:

1. `oshcc`: for compiling and linking C programs.
2. `oshCC`: for compiling and linking C++ programs.
3. `oshfort`: for compiling and linking F77/F90 programs.
4. `oshrun`: to launch programs.

The similarity to the style of wrappers found in many MPI implementations is obvious and intentional. Currently these wrappers do handle a few situations (e.g. oshcc/CC and oshfort detect that they shouldn't do linking and stop the underlying compiler complaining about link options being present, but unused). The compiler scripts are generated from a common template.

The run wrapper currently detects which GASNet conduit is being used and sets up the environment accordingly to launch the program. Not sure if this is the best place to do this check, or if the build process should work this out in advance to streamline the installed code.

11 Configuration and Installation

There is a `trunk/configure` script that is a simplified version of the GNU autotools. This script will eventually become the GNU setup and will do lots more feature tests. So the usual procedure applies: configure, make, make install.

12 Future Plans

Ideas for extensions to SHMEM to go into OpenSHMEM need to be requested and evaluated from the SHMEM user and vendor community. A decision process will determine which ideas are eventually implemented. The library that this document refers to is hopefully a good platform for these developments.

A number of extensions have already been proposed, and in fact have been implemented in other SHMEM libraries. These include (but are not limited to)

- thread-safety: providing thread-safe SHMEM routines that can operate in threaded environments, *e.g.* alongside OpenMP;

- non-blocking puts: put routines that return per-communication handles to the caller. The handles can be tested later for completion (present in Cray and Quadrics SHMEMs); this extension may require revamping the way implicit handles are used in GASNet, since we will be generating calls with explicitly generated handles. Building a handle pool on which to synchronize later should take care of this.

- locality: exposing information about topology to the library and/or its API;

- regularized namespace: currently routines are a strange brew of "shmem_" prefixes, "start_pes", "_my_pe" and so on. Providing an API with a consistent naming scheme would be useful;

- Fortran module, C++ API: provide better language support.