Thread-safe SHMEM Extensions

CRAY

October 2012

Abstract

This document is intended to serve as a proposal for thread safety extensions to the OpenSHMEM specification and at the same time describes planned support for thread-safety for Cray SHMEM on certain Cray systems.

Introduction

The original impetus for implementing thread-safe Cray SHMEM and proposing thread safety extensions to OpenSHMEM were requests from SHMEM customers for thread safety support. Subsequent discussions with some SHMEM customers and review of the MPI specification led to the proposal detailed in this document. The document describes basic thread safety support for Cray SHMEM. The thread safety support is basic in that it imposes policies on SHMEM applications and contains minimal extensions to the Cray SHMEM and OpenSHMEM APIs. However, it does enable processes to issue small Puts, small Gets, and AMOs at higher aggregate rates than is possible in a single-threaded environment, which leads to better performance for certain multi-threaded applications. As much as possible, this proposal was guided by the thread safety extensions to the MPI standard and by customer input, in an effort to facilitate acceptance by the OpenSHMEM community.

Note that what is known in a single-threaded environment as a processing element (PE) or rank corresponds to a process, not a thread, in a multi-threaded environment.

The remainder of the document will discuss the proposed thread safety extensions to OpenSHMEM, including assumptions and new functions.

Initial Assumptions

As mentioned above, discussions with some SHMEM customers and review of the MPI specification led to this proposal, including several agreed-upon assumptions.

1. Thread safety support is required for Put, Get and AMO operations.
2. SHMEM collectives operate on sets of processes and the use of SHMEM collective calls with multiple threads per process can be problematic. Thus, SHMEM collective calls are subject to the following restrictions:
    a. A collective operation can be called from only one thread per process at a time and several threads per process cannot simultaneously participate in different collective operations.
    b. Collective operations must be protected with thread barriers calls before and after each collective call, i.e. a SHMEM application developer must implement the following steps:
        - ○ On each process, all threads that make SHMEM calls participate in a local barrier.
        - ○ One thread per process participates in the collective operation.
        - ○ On each process, all threads that make SHMEM calls participate in a local barrier.
3. Initialization and finalization routines are restricted to being called by one thread per process. A new initialization routine, **shmem_init_thread()**, enables the user to specify that support for thread safety is desired.

4. The symmetric heap management functions **shmalloc()**, **shrealloc()**, and **shfree()** are all defined to call **shmem_barrier_all()** before they return and thus must be treated as collective operations.

5. The lock functions **shmem_clear_lock()**, **shmem_set_lock()**, and **shmem_test_lock()** are restricted such that multiple threads on the same process cannot access the same lock at the same time. Note that this restriction does permit two different threads on the same process to access two different locks at the same time.

6. Cray will propose the thread safety extensions described in this document to the OpenSHMEM committee for inclusion in the OpenSHMEM standard.

Precautions

Programmers using thread-safe SHMEM should be mindful of the following caveats.

1. No thread safety locking is added to collective operations, including barriers, lock operations, and **shmalloc()**, **shrealloc()**, and **shfree()**. Provided programmers observe the above-mentioned restrictions, thread safety locking is unnecessary and would only slow program execution.

2. The SHMEM programming model does not recognize individual threads. Any SHMEM operation initiated by a thread is considered an action of the process as a whole. In particular, note that:

    a. **shmem_quiet()** and **shmem_fence()** affect the entire process, not just the calling thread. Depending on implementation, a call to **shmem_quiet()** by one thread may or may not prevent other threads from simultaneously making SHMEM calls (see 3. below).

    b. The symmetric heap is a per process resource. A thread making a **shmalloc()**, **shrealloc()**, or **shfree()** call affects the entire process. The existing requirement that the same symmetric heap operations must be executed by all processes in the same order also applies in a multi-threaded environment.

3. When using multiple threads and SHMEM, be mindful of the order of access and race conditions. For example, if one thread of a process is issuing Puts and another thread of the same process is calling **shmem_quiet(),** it is the programmer's responsibility to ensure the correct ordering of those operations.

4. Thread safety should not be activated unless needed. Activating thread safety causes additional overhead even if no additional threads are created or used.

SHMEM Thread Safety Extensions

Where appropriate, SHMEM thread safety extensions have been modeled after the exiting MPI thread safety interface. The following naming convention applies: functions which relate to the level of thread safety activated are named **shmem_<action>_thread()**. Functions which apply to a specific thread only are named **shmem_thread_<action>()**. Return types of new functions are chosen to follow the OpenSHMEM model of being void or returning a result. This differs from the MPI model where functions return a success or failure code and results are passed via output parameters.

shmem_init_thread()

A new function, **shmem_init_thread()**, allows a user to indicate that thread safety support is desired. The function initializes SHMEM in the same way that **shmem_init()** does. In addition, it performs thread safety specific initialization. This function is used in place of **shmem_init()** either before additional threads are created or by only one thread per process. The thread which calls **shmem_init_thread()** is known as the main/primary thread. The syntax of the function is as follows:

```
int shmem_init_thread(int required)
```

There are four levels of threading support. Note that these levels are hierarchical. The higher levels should support any lower levels.

1. SHMEM_THREAD_SINGLE -- no threading/one thread per process. SHMEM implementers can assume there is no threading.
2. SHMEM_THREAD_FUNNELED -- processes may have multiple threads but only one of the threads can make SHMEM calls. (All functions are funneled through one thread.) It is the user's responsibility to make certain all SHMEM calls by a process are executed by the same thread.

   Cray does not provide support for this level. It is unclear whether this level should be included in the OpenSHMEM spec.
3. SHMEM_THREAD_SERIALIZED -- processes may have multiple threads. Any thread may issue SHMEM calls, but only one SHMEM call per process can be active at any given time. Simultaneous calls from two threads belonging to the same process are not allowed. It is the user's responsibility to make certain that SHMEM calls by a process are not concurrent.

   Cray does not provide support for this level. It is unclear whether this level should be included in the OpenSHMEM spec.
4. SHMEM_THREAD_MULTIPLE – processes may have multiple threads. Any thread may issue a SHMEM call at any time, subject to the restrictions and policies described earlier.

To specify which level of threading support is desired, use the **shmem_init_thread()**'s "required" argument to pass in one of the above symbols, specifying which level of threading support is desired. The return value of the function is the level of threading support that the SHMEM library can provide.

If possible, the function returns the "required" symbol. If that is not possible, the library returns the lowest threading support level that can be supported that is greater than "required." If that is not possible, the function returns the highest threading support level SHMEM can provide. The user is responsible for checking the return value to make certain that the available thread-safety level is suitable for their program.

All processes in a SHMEM application must request the same level of threading support.

It may turn out that additional input parameters should be added to shmem_init_thread(), e.g. to allow specification of a maximum concurency level. This will be reviewed at a later time.

Calls to the standard SHMEM initialization routines, **shmem_init()** and **start_pes()**, are considered to request the threading support level SHMEM_THREAD_SINGLE.

shmem_query_thread()

A new query function, **shmem_query_thread()**, enables SHMEM application developers to query the current level of thread safety support. When invoked, it returns the same thread safety level that was returned when **shmem_init_thread()** was called.

The syntax is as follows:

```
int shmem_query_thread(void)
```

shmem_thread_register()

After **shmem_init_thread()** has been called by the primary thread, any thread, including the primary thread, that wishes to make SHMEM calls must call **shmem_thread_register()** before making any other SHMEM calls. Should the primary thread need to call shmem_thread_register(), it can do so before starting or after having started the other threads. An error may be returned if the maximum level of concurrency is exceeded. Final decisions regarding return type and input/output parameters remain to be made.

The syntax is as follows:

```
int shmem_thread_register(int *params)
```

shmem_thread_unregister()

Any thread that previously called **shmem_thread_register()** must call **shmem_thread_unregister()** before exiting. For all such threads but one this must be the last call to SHMEM. One thread will call **shmem_thread_unregister()** followed by **shmem_finalize()** (see Initial Assumptions). Details regarding return type and input/output parameters remain to be made.

The syntax is as follows:

```
int shmem_thread_unregister(int *params)
```

The following pseudo code illustrates a sample call sequence at the end of a multi-threaded SHMEM program.

```
if (not_primary)

    shmem_thread_unregister();

pthread_barrier;

if (primary)
{

    shfree();

    shmem_thread_unregister();

    shmem_finalize();

}
```

shmem_thread_quiet()

This is the thread specific version of the **shmem_quiet()** function. It allows an individual thread to wait for completion of Puts and non-blocking Gets which it previously issued. There is no requirement on the implementation to only complete operations issued by the calling thread. Final decisions regarding return type and input/output parameters remain to be made.

The syntax is as follows:

```
int shmem_thread_quiet(int *params)
```

shmem_thread_fence()

This is the thread specific version of the **shmem_fence()** function. It allows an individual thread to ensure ordering of Puts and non-blocking Gets which it previously issued. There is no requirement on the implementation to only order operations issued by the calling thread. Final decisions regarding return type and input/output parameters remain to be made.

The syntax is as follows:

```
int shmem_thread_fence(int *params)
```

Future Enhancements

The proposed thread safety extensions are modeled after the MPI standard for thread safety, as it is hoped that by following a well-known and well-defined interface, the proposed extensions will be more readily accepted. The thread-safe SHMEM interface could be expanded beyond the minimum required to include functions such as **shmem_thread_is_main(), shmem_thread_barrier()** etc., should this be desired by SHMEM users in the future. Some of the policies imposed on SHMEM applications may be lifted over time.