Thread-safe SHMEM Extensions

Monika ten Bruggencate[1], Duncan Roweth[1], Steve Oyanagi[1]

Cray Inc.

Abstract. This paper 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 Cray XE and XC systems.

1 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 [1] led to the proposal detailed in this paper. The paper 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 can lead 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 paper will discuss the proposed thread safety extensions to OpenSHMEM, including assumptions and new functions.

2 Assumptions

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

1. 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.
2. Thread safety support is required for Put, Get and AMO operations so that an application with multiple threads per process can make one-sided SHMEM calls from multiple threads.

3. Not necessarily all threads make SHMEM calls. It may be that only a subset of the threads of a process make SHMEM calls.
4. The pool of threads that make SHMEM calls may be static or may be dynamic.
5. SHMEM collectives operate on sets of processes and the use of SHMEM collective calls with multiple threads per process can be problematic. For instance, in the OpenSHMEM specification 1.0 [2], in some cases collective operations are defined not only in terms of PE synchronization, but also memory consistency. The semantics of operations like shmem_barrier() would either need to be redefined for the proposed thread-safety extensions, or new SHMEM collective functions would need to be defined which would address these memory consistency issues. Either effort is beyond the scope of this initial proposal and should be coordinated with future proposals for modernizing the collective operations component of the SHMEM API. Thus, for the purpose of this paper, 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) Where an application makes SHMEM collective calls from multiple threads per process, it is the responsibility of the application to ensure that calls are made in the same order in each participating process.
6. The symmetric heap management functions `shmalloc()`, `shfree()`, and `shrealloc()`, and `shfree()` are all defined to call `shmem_barrier_all()` before they return and thus must be treated as collective operations.
7. 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.
8. Cray is proposing the thread safety extensions described in this paper to the OpenSHMEM committee for inclusion in the OpenSHMEM standard.

3 Precautions

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

1. It is the applications responsibility to ensure that collectives are called in the right order, no matter whether the application is single-threaded or multi-threaded. 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. While a thread is calling `shmem_quiet()` or `shmem_fence()`, no other thread should be able to make calls who's behavior is affected

by `shmem_quiet()/shmem_fence()`. Further, a call to `shmem_quiet()` by one thread should affect all threads in that the call will wait for completion of all outstanding Puts and non-blocking Gets issued by the process. Similarly, a call to `shmem_fence()` by one thread should affect all threads.

(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.

2. 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.
3. Thread safety should not be activated unless needed. Activating thread safety causes additional overhead even if no additional threads are created or used.

4 SHMEM Thread Safety Extensions

Where appropriate, SHMEM thread safety extensions have been modeled after the existing 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.

4.1 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()` or `start_pes()` 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, int max_num_threads)
```

Following the MPI standard, there could be 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` value. 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.
The input parameter `max_num_threads` allows a user to specify the maximum number of threads per process that will make SHMEM calls. If the maximum number is not known, a negative number (or appropriate macro) indicates that no upper limit exists. Knowing the maximum number allows lower level software to optimize use of hardware and minimize startup and teardown overhead.
Additional input parameters may be added to `shmem_init_thread()` as our implementation progresses. Calls to the standard SHMEM initialization routines, `shmem_init()` and `start_pes()`, are considered to request the threading support level `SHMEM_THREAD_SINGLE`.

4.2 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)
```

4.3 shmem_thread_register()

After `shmem_init_thread()` has been called by the primary thread, any other thread that wishes to make SHMEM calls must call `shmem_thread_register()` before making any other SHMEM calls. The primary thread which called `shmem_init_thread()` does not have to call `shmem_thread_register()`. Introducing a function which explicitly registers threads has several advantages.

- It allows optimized use of hardware, for instance by initializing thread safe storage to allow dedicated use of hardware components by a thread.
- It minimizes overhead of pt2pt operations since such operations don't have to check a thread-private registration variable and, if necessary, perform thread registration under the cover.
- It allows dynamic creation and destruction of threads throughout a program run while not wasting hardware resources. For instance, if a thread is destroyed in the middle of a program run and calls `shmem_thread_unregister()` prior to that event, the hardware resources which were dedicated to that thread can be reassigned to another thread which calls `shmem_thread_register()` later in the program run.

An error may be returned if the maximum level of concurrency is exceeded, i.e. if more threads are attempting to register than was specified via the `max_num_threads` input parameter to `shmem_init_thread()`. The syntax is as follows:

```
int shmem_thread_register(void)
```

4.4 shmem_thread_unregister()

Any thread that previously called `shmem_thread_register()` must call `shmem_thread_unregister()` before exiting. The thread that called `shmem_init_thread()` does not have to call `shmem_thread_unregister`. The syntax is as follows:

```
int shmem_thread_unregister(void)
```

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

```
if (primary) {
shmem_init_thread()
pthread create loop
shmem calls
pthread join loop
shmem_finalize()
}
```

```
if (non-primary)
{
shmem_thread_register()
shmem calls
shmem_thread_unregister()
}
```

4.5 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. The syntax is as follows:

```
void shmem_thread_quiet(void)
```

4.6 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. The syntax is as follows:

```
void shmem_thread_fence(void)
```

4.7 shmem_thread_barrier()

This function performs an efficient local barrier among the threads that have registered themselves by calling `shmem_thread_register()`. Decisions regarding return type and input/output parameters remain to be made. The syntax is as follows:

```
void shmem_thread_barrier(void)
```

4.8 shmem_thread_is_registered()

This function determines whether a thread can make SHMEM calls. It will return TRUE if the thread has previously called `shmem_thread_register()` or is the main thread. It will return FALSE otherwise. The syntax is as follows:

```
int shmem_thread_is_registered(void)
```

4.9 Sample pseudo code

The following pseudo code illustrates a possible code flow and usage of the new API functions. It is not related to any real world example.

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

int
main (int argc, char *argv[])
{
    int nthreads, /* number of threads */
        tid,      /* thread id */
        rc;       /* return value */

    Initialization phase;

    rc = shmem_init_thread(SHMEM_THREAD_MULTIPLE, 8);

#pragma omp parallel private(tid) /* fork threads whith each having a private tid */
    {
        if (tid != 0)
            shmem_thread_register(); /* additional threads must register  */

        shmem_int_get_nb(...);  /* all threads transfer data */

        Computation phase by each thread;

        if (tid == 1)
            shmem_thread_quiet();  /* thread 1 waits for own transfer completion */

        Computation phase by each thread;

        if (tid == 0)
            shmem_quiet();  /* wait for transfers of all threads to complete */

        Computation phase by each thread;

        shmem_thread_barrier();  /* synchronize all threads of the rank */

        if (tid == 2)
            shmem_barrier(...);  /* one thread participates in a collective */

        if (tid != 0)
            shmem_thread_unregister(); /* additional threads unregister */

    }  /* all threads join master thread and terminate */

    shmem_finalize();
}
```

5 Performance Considerations

The original impetus for supporting thread-safe Cray SHMEM came from customer requests, where requests focused on point-to-point operations and on performance. Thus, one goal when implementing the described thread safety extensions on Cray XE and XC systems is to increase the per process, aggregate issue rate for Puts, Gets and AMOs. At the time of the publication of this paper, the implementation of the thread safety extensions in Cray SHMEM has not been completed and we are not able to present performance data at the SHMEM level. In our software stack, Cray SHMEM is implemented on top of DMAPP, a network library supporting one-sided program models. The majority of the work to support thread safety in a well-performing manner needed to occur in DMAPP and has been completed. DMAPP was modified to use NIC resources effectively in the presence of threads. Specifically, registering a thread with DMAPP allows the thread to use a dedicated NIC resource, thereby eliminating bottlenecks when accessing hardware resources. This approach also allowed us to eliminate the use of a global lock in DMAPP. We carried out preliminary performance experiments at the DMAPP level, comparing the older implementation where a thread did not use a dedicated NIC resource and which used a global lock, with the new, better-performing implementation. Preliminary performance data on XE shows an increase in per-process aggregate issue rate by a factor of 4 to 5 for non-blocking 8 byte Puts using 8 threads per process. More thorough performance analysis will be done at the DMAPP level and, once the work has been completed in Cray SHMEM, at the SHMEM level. We will then use the performance analysis to guide further improvements in our software stack.

6 Future Work

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 should this be desired by SHMEM users. Some of the policies imposed on SHMEM applications may be lifted over time. In particular, we plan to work with the community to determine whether and how policies on the use of collective operations in a multi-threaded environment should be loosened over time.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Standard Version 3.0. (2012)
2. OpenSHMEM: OpenSHMEM Specification v1.0. (2012)