A Multithreaded Communication Substrate for OpenSHMEM

Aurelien Bouteiller, Thomas Herault, George Bosilca
Innovative Computing Laboratory
The University of Tennessee

ABSTRACT

OpenSHMEM scalability is strongly dependent on the capability of its communication layer to efficiently handle multiple threads. In this paper, we present an early evaluation of the thread safety specification in the Unified Common Communication Substrate (UCCS) employed in OpenSHMEM. Results demonstrate that thread safety can be provided at an acceptable cost and can improve efficiency for some operations, compared to serializing communication.

1. INTRODUCTION

MPI (Message Passing Interface) has served the High Performance Computing community extremely well during the last two decades. So well in fact that it has been promoted from its design intent as a communication library to, in effect, a *programming paradigm*. However, as the gap between the assumptions of the MPI programming paradigm and the hardware capabilities of machines widens, the pressure to switch toward innovative programming approaches increases. OpenSHMEM [4] is such an approach that is recently enjoying a revival in interest, thanks to its flexible ability to express complex irregular memory accesses (henceforth irregular communication patterns), which are becoming increasingly pervasive in scientific applications designed to scale on clusters of hybridized, complex processors.

One crucial capability for OpenSHMEM is to benefit from high performance communications to resolve remote memory accesses in a timely and efficient manner; although MPI could be employed to that end [3], its two-sided primitives poorly fit with the intrinsically one-sided operations exposed to the user in OpenSHMEM. Meanwhile, one-sided MPI primitives impose more synchrony than necessary, thereby reducing potential performance.

The Unified Common Communication Substrate (UCCS [4]) is a lower level communication layer that features routines matching both the MPI and OpenSHMEM communication models. In this paper, we present the design and an early performance analysis of the thread safety support in UCCS. The key contributions are 1) the extension of the UCCS specification to support multiple threads, 2) the creation of UCCS benchmarks measuring thread management efficiency, and 3) to demonstrate that thread safety can be achieved at a reasonable cost for the OpenSHMEM communication layer.

The remainder of this paper is organized as follows: in Section 2 we present the key features of the UCCS thread safety design; In Section 3 we outline early performance results; and then we conclude in Section 4.

2. UCCS THREAD SAFETY DESIGN

Runtime Environment. The UCCS specification includes a portable API to abstract Runtime Environment (RTE) services. The RTE is responsible for the deployment, I/O forwarding, and the exchange of the network identity cards during connection setup. Unfortunately, many popular RTEs deployed in production have limited or non-existent thread safety. Since the RTE API is used only during the establishment of new connexions and not during performance critical communication routines, our implementation choice is to delegate the RTE progress to a library internal thread, and to serialize all RTE service calls with a global mutex, taken in the RTE API shim routines. This ensures thread-safe access to an unmodified non-thread safe RTE infrastructure.

Progress. On the contrary, the UCCS communication library implementation does not feature an internal progress thread. The API is designed to allow simultaneous calls from multiple user threads, resulting in multiple threads entering the UCCS progress loop, and making these threads available to the UCCS engine to perform background progress, if necessary (as an example to sustain OpenSHMEM asynchronous progress). Despite a careful design of the UCCS library internals, this flexible design means that at any time, multiple threads could concurrently manipulate internal objects, such as message queues and requests. We rely on some of the highly optimized atomic operations and data structures from OPAL [2] to provide the fine grain locking and thread safe, and possibly lockless, accessors necessary to implement requests management and message queues (as had been hinted as most efficient in the context of MPI [1]).

Posting operations. Similarly, API calls to initiate non-blocking operations can be invoked from multiple threads simultaneously. Our general design encourages an imple-

mentation with fine grain locking, to ensure minimal thread contention. The current implementation of most operations (one-sided PUT, GET) is indeed implemented with per-request/endpoint locking; however, the emission of active message currently rely on mutex serialization. While this is outside the scope of the current thread safety analysis, there is an ongoing effort to investigate different approaches of further minimizing the overheads due to synchronizations in the critical path, an effort mirrored closely at the OpenSHMEM community with two active threading proposals (endpoints and contexts).

Active Message callbacks. Active Messages (AM) are a powerful construct to implement efficient high level communication protocols. On processes that need to act on the reception of a message, a tag and a callback are registered. Upon reception of a tagged message, the callback function associated with the tag is triggered. In the context of a multithreaded application, the rules regarding which threads can execute the callback must be explicit. In the UCCS specification, we decided to allow any thread to execute any callback, even when the callback has been initially registered by another thread. Furthermore, if multiple active messages with the same tag are incoming, multiple instances of the callback may coexist simultaneously, in which case, mutual exclusion for any global structure accesses through the callback is the user's responsibility.

3. PERFORMANCE EVALUATION

Performance evaluation is carried on two nodes of an Infiniband(20G) cluster. Each node features two quad-cores Intel Xeon E5520 (Nehalem) with 12GB of DDR3 memory. Software environment is Linux CentOS 6.5. In this section, we also introduce original UCCS multithreaded synthetic benchmarks, inspired from MPI benchmarks [5].

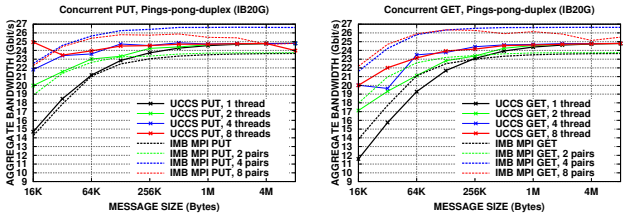

Figure 1: Aggregate bandwidth of one-sided communication primitives, issued by multiple threads concurrently.

The first experiment (Figure 1) compares the bandwidth of one-sided operations when an increasing number of threads are posting concurrent communication requests. In this benchmark, each process spawns the demanded number of threads that post concurrent PUT and GET operations (left and right graphs respectively) representing a number of "Pings". A supplementary thread polls the destination memory locations to detect the "Ping" transfer completion and then emits a single short 4 bytes "Pong" message. Both processes initiate a Ping wave, thereby measuring duplex bandwidth. With one thread, the observed maximum bandwidth with UCCS is superior to (non thread-safe) MPI performance (IMB 4.0, bidirectional RMA aggregate mode, Open MPI 1.7.5). Both PUT and GET maximum bandwidth are similar, but the bandwidth for intermediate message size is higher for PUT than GET operations. The addition of a second thread performing concurrent communication increases the observed bandwidth (especially for medium size messages), closely matching multiple MPI processes performing the same communication pattern. When the number of threads is further increased to 4 and 8, the UCCS bandwidth still increases for medium messages, but cannot match anymore the best bandwidth obtained with MPI (with 4 processes per node).

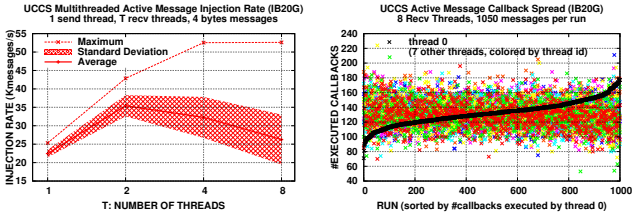

Figure 2: Injection rate and load fairness for Active Message communications with multiple receiver threads.

The second experiment (Figure 2) investigates Active Message callbacks processing. In this benchmark, a single thread on the sender process emits a large number of 4 bytes AM requests. The receiver process registers a minimalistic AM callback to count the number of incoming messages and send an ACK when a threshold is met. The reception of the ACK permits computing an estimate of the injection rate at the sender. In all multithreaded cases, the performance is improved compared to the single thread runs (up to 75% better average injection rate when employing 2 threads). The widening gap between the maximum and the average injection rates indicates some remaining contention. The right graph presents the spread of the callbacks among 8 receiver threads for 1000 runs. As can be observed, the spread is generally fair with an identical average and a low standard deviation for each thread.

4. CONCLUSIONS

We presented early results regarding multi-thread support in the UCCS specification. The Implementation experience and performance demonstrate that the design has the potential to deliver bandwidth and fairness when multiple threads are concurrently communicating. In future works, we intend to eliminate the remaining coarse grain mutex and expand the benchmarks to include overlap, network congestions and realistic OpenSHMEM application workloads.

5. REFERENCES

[1] P. Balaji, D. Buntinas, D. Goodell, W. D. Gropp, and R. Thakur. Fine-grained multithreading support for hybrid threaded MPI programming. *Int. J. High Perform. Comput.Appl.*, 24:49–57, Feb. 2010.

[2] E. Gabriel, G. E. Fagg, G. Bosilca, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. volume 3241 of *LNCS*, pages 97–104, 2004.

[3] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing OpenSHMEM using MPI-3 one-sided communication. volume 8356 of *LNCS*, pages 44–58. Springer, 2014.

[4] P. Shamis, M. G. Venkata, S. W. Poole, A. Welch, and T. Curtis. Designing a high performance OpenSHMEM implementation using universal common communication substrate as a communication middleware. volume 8356 of *LNCS*, pages 1–13. Springer, 2014.

[5] R. Thakur and W. D. Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35:608–617, Nov. 2008.