SAND2015-8650C

Lightweight threading with MPI using Persistent Communications Semantics

Ryan E. Grant[*]
Sandia National Laboratories
Center for Computational Research
P.O. Box 5800, MS-1110
Albuquerque, NM 87185-1110
regrant@sandia.gov

Anthony Skjellum
Auburn University
Department of Computer Science and Software Engineering
Auburn, AL 36849
skjellum@auburn.edu

Purushotham V. Bangalore
University of Alabama at Birmingham
Department of Computer and Information Sciences
Birmingham, AL 35294
puri@uab.edu

ABSTRACT

Multi-threaded performance in MPI is of concern for future systems, particularly at Exascale, where massive concurrency will be necessary to leverage the full power of systems. While MPI provides generalized solutions and additional proposals like endpoints expand this general model, examining common use cases that have good solutions that may not be universally applicable is a viable additional approach. This paper details a new conceptual concurrency support mechanism for MPI that is applicable to a (large) subset of MPI applications. This approach is expected to provide very low overhead while still allowing for optimizations in the MPI library that are not currently possible.

1. INTRODUCTION

The Message Passing Interface (MPI) [6] has supported threaded execution in user applications since Version 2 [4] was released in 1997. To this day, MPI multi-threading support remains not well optimized. It has been hard for thread-based code to interact with MPI and many hybrid codes today do not allow for thread interaction with MPI. For example, an MPI/OpenMP hybrid code might perform a computation using many threads in a loop, but then use a single master process to communicate using MPI. It is desirable for threaded codes to interact with MPI, but many codes do not actually require the full MPI multi-threaded support that exists today. The current commonly used MPI implementations *pessimistically* use locks to enforce thread safety to MPI calls (*e.g.,* serialization), allowing only a single thread to be interacting within certain MPI critical paths at a given time. While this may be desirable for a general threading case where no particular behavior of threads is guaranteed, it can be heavyweight for threads that would otherwise not interfere with each other in their participation in a communication (that also lacked ordering constraints). For example, if multiple threads each wrote to a shared memory buffer to assigned locations based on offsets related to their thread IDs, no interference would occur between these threads. When sending the buffer to the target, that buffer can be sent after all of the threads have finished, or separate portions could be sent in a MPI_Send calls.

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The overheads for imposing thread safety for MPI have been studied [8], and most implementations as of 2015 do not have sophisticated locking mechanisms for enforcing thread safety. Some approaches simply use a single global lock on the MPI library, while others try to lock at a fine-granularity. When using blocking MPI operations, there is a serialization of the data flowing out of the threads over the wire, while there may be no inherent need to enforce this ordering.

Multi-threading support for MPI has traditionally focused on providing thread-safety, with multiple different operating modes for supporting thread-based concurrency. All of these modes concentrate on the methods by which MPI can arbitrate between threads and what restrictions it must put in place to ensure thread-safety. The approach presented in this paper places the burden of managing safe threading on the application, which has good information on which to base its approach to concurrency.

2. LIGHTWEIGHT CONCURRENCY

Solutions to multi-threading in MPI such as Endpoints [3], provide for parallelism with threads through exposing the threads as additional ranks (or some application-prescribed subsets thereof). For some common use cases, endpoints are overly complex. They lead to increased application code complexity as well as greater network resource usage (because of greatly expanded rank space, and concomitant state). Application code complexity and the amount of change that existing codes must endure in order to leverage thread parallelism in future codes is of great concern. Multi-million line MPI codes require great effort to modify (and revalidate) and therefore minimal code change and familiarity to existing multi-threaded approaches is desirable for efficiency in both time and cost for updating legacy code bases.

2.1 A Solution for Many Applications

Existing multi-threaded code can be written such that many threads (actors) contribute to a single solution in a shared memory buffer. The need for each of the threads to have knowledge of each other is limited, with the basic knowledge of where the solution must be written to for each thread being of primary concern. As such, the need for thread to thread communication in MPI is concomitantly limited, the access to the larger input buffer of data used and the output buffer for the solution is the main concern. For example, for a stencil code, one might further break up the simulation space within a given MPI process in the same way that it was done for the multiple processes themselves. The data to be exchanged (a face of a 3D simulation space) with the process neighbor only needs the face data for the process. It does not need to understand the number of threads nor the layout of the data that each of the threads is working on in another process. What is needed is that the entire face data be sent to the other neighbor process.

To enable this type of computation/communication in MPI with minimal overheads, the threads can deliver their portion of the overall data to MPI with an MPI_Partitioned_Add_to_buffer[1] call. The intention of this proposed function is to provide a portion of data to an operation that will collect many pieces of data from many actors and deliver the payload to the requested MPI process. This approach requires that some information about the partitioned operation be expressed to MPI prior to writing to any buffers. The partitioned operation interface leverages the persistent communications interface in MPI to provide this data. First, the operation must be initialized; this will provide the required information to setup the buffers and the synchronization methods (which could be as simple as an atomic increment on a counter). This operation can subsequently be started and finished as a normal persistent communication would be, using the same semantics as a traditional persistent communication, with modifications for the partitioned nature of the operation. An MPI_Partitioned_Send_Create(Comm, to_rank, to_tag, base_address, data_type, count, num_contributors, &request)[2] call can be used to initialize the partitioned send, which is similar to a persistent operation setup. The mechanisms used to start/stop the operation are similar to persistent operations as well. Calling a MPI_Start(request) call will activate the partitioned send, but the actual data transmission will only start/complete once the parts of the message are delivered to MPI. The most naïve implementation of this would be that no data be sent over the wire until all parts of the overall communication had been assembled by MPI (of course parts of the message could be sent before all parts are received for optimization purposes, as will be discussed later). The method of finishing a partitioned send is a simple MPI_Wait(request) as one would typically wait for a request to complete in MPI. This process is illustrated at a high level in Figure 1. It is important to note that in this paper we refer to the actors on the buffer as threads, when they could be tasks instead.

While similar communication could be accomplished with many individual smaller MPI_Send calls from each thread, this partitioned data approach has several key advantages. First, the overhead of synchronization between threads can

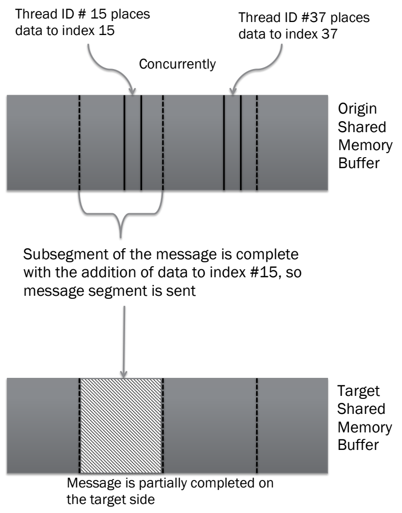
Thread ID # 15 places data to index 15
Thread ID #37 places data to index 37
Concurrently
Origin Shared Memory Buffer
Subsegment of the message is complete with the addition of data to index #15, so message segment is sent
Target Shared Memory Buffer
Message is partially completed on the target side

Figure 1: An example of multiple threads placing data into a partitioned buffer with partial buffer sending capabilities in the MPI library.

be reduced, inasmuch as each contributor to the single larger MPI operation can add its respective data and all that is needed is an atomic increment to maintain the count on the number of contributions to the partitioned send (the number of contributions is known at the point that the partitioned operation is initialized). This has much lower overhead than the current MPI_THREAD_MULTIPLE methods for ensuring thread-safe MPI operations, as it does not require locking of key MPI library functions as the impact of the call is confined to the partitioned operation's buffer[3]. In addition, there are opportunities for the MPI library to provide optimizations to the communication as a whole. For example, a partitioned send could take one of two extremes in when it would place data out on the wire for transmission, it could send the data as one large message once all of the parts have been placed in the buffer. Alternatively, it could send each individual part as they are placed. Of course, any combination in between these two extremes could also be implemented. This can provide certain benefits, particularly when knowledge about the network can be applied, like sending message chunks that are the size of the underlying network MTU whenever they are complete. This should provide a steady lower bandwidth requirement communication stream, that would also minimize wire-side communication overhead by optimizing payload sizes with respect to header/tail data. While this can also be accomplished by sending large messages, this approach can lead to less bursty traffic over time, lessening the possibility of temporally localized stress on the network resources.

2.2 Applicability to Stencil Codes

Stencil codes can be satisfied in their threading requirements with a call to MPI that allows for partial placement of data into a communication buffer, letting MPI pack the larger communication buffer as a whole and communicating the entirety of it to other MPI processes. Since this threading/communication system relies on underlying shared memory among the threads, no scatter operation needs to take place on the receiving process, so long as the target for place-

[1]A new proposed API by the authors.

[2]Additional new API functionality.

[3]Furthermore it works to localize any additional overhead

ment of the data is a shared memory window. This also leverages all of the potential performance optimizations that could be implemented as previously discussed. For multithreading, MPI is providing a method of operation that has already existed outside of MPI for some time. It is notably important to utilize the current paradigms in threaded programming with MPI proposals, to aid in an easier transition to multithreaded MPI code.

2.3 Flexibility Through Expressive Inputs

A challenge in further meeting the needs of application codes is typically encountered in the initialization phase. This challenge is that codes may need to place non-contiguous partitioned data, but may be able to do so in a predefined pattern. In order to support multiple common needs with respect to data placement, two approaches can cover the majority of requirements. The first is the simpler of the two, providing a bit mask, which determines where in the buffer as a whole each of the provided items should be placed. The function MPI_Partitioned_Add_to_buffer(request, in_data, in_datatype, num_contributions, mask[]) can be used for this purpose. However, for cases where there are a large number of contributions to be made or the buffer is huge, the bitmask approach becomes unwieldy, because the size of the bitmask can be significant and lead to performance and memory space utilization issues. Consequently, it is useful to offer an alternative approach, in which the data placement can be expressed in a vector. This lowers both the memory requirements for expressing where the data should be placed as well as the issues surrounding passing in a large mask. However, this is more complicated for the application programmer to use and may be more difficult to process in the MPI library for small partitioned buffers.

2.4 Hardware Support for Partitioned Send

Dedicated hardware support for partitioned send operations is unnecessary if the networking hardware provides some basic building blocks on which the operations can be built. An example of a networking solution that can support partitioned sends today is Bull's BXI interconnect [2]. Bull's BXI network uses the Portals 4 networking API [1], which supports triggered operations. Triggered operations use a hardware counter on the networking devices to accumulate counts of certain events that can be associated with them. Consequently, on the receive side, a Portals-compatible NIC can keep a count of the number of expected contributions to a buffer and deliver notification of the completion to the target immediately upon completion of the operation. The send-side MPI library can leverage triggered operations as well, by staging multiple requested send operations with different counts on which they are triggered. Using the PtlCTInc function in the Portals 4 API, MPI can keep the bookkeeping required for subsections of the partitioned buffer on the NIC hardware. Once a given sub-partition of the overall buffer has been placed, the hardware automatically triggers the send to occur. This allows for increased network efficiency while offloading a large portion of the work that would otherwise have to occur in software (counting incoming segments and determining when a request was complete).

3. RELATED WORK

There have been past attempts to integrate threading within MPI, like MPI/RT [7] and FG-MPI [5] and, while promising, have not become widely used MPI implementations and have not been integrated into the MPI standard. Work has been done in analyzing the performance of existing threading modes in the standard [8]. The MPI forum, in 2015, has a proposal before it to support threads through endpoints [3], in which each thread can be assigned a unique rank in a endpoint communicator. Both the current threading support in MPI and the endpoints proposal require that MPI be able to manage individual threads either through thread safety or by using additional resources to account for the threads (resp, endpoints). This work is differentiated from previous efforts insofar as the requirements it places on the applications, and the corresponding decrease both in resources needed by MPI and synchronization overhead.

4. CONCLUSIONS AND FUTURE WORK

The partitioned send approach has been introduced and the key motivations behind the need for such an approach have been discussed. Providing threading support through implicit methods such as persistent communication type partitioned sends allows for low overhead thread-safety as well as fitting the existing application code methodologies. Although this approach may not work for every possible use case for threading support, it does provide a solution to a substantial portion of the overall problem space. Therefore, it holds promise as a potential methodology for future application codes to enable threading support while maintaining high performance network communication.

Future work in this area will involve the creation of a prototype partitioned send/recv. We intend to test this approach with several different proxy applications of interest to determine the performance benefits when compared to MPI_THREAD_MULTIPLE operation.

5. REFERENCES

[1] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson. The Portals 4.0.2 networking programming interface, 2014.

[2] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. Wellenreiter. The BXI interconnect architecture. In *Proceedings of the 23rd Annual Symposium on High Performance Interconnects*, HOTI '15. IEEE, 2015.

[3] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible mpi endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[4] S. Huss-Lederman, B. Gropp, A. Skjellum, A. Lumsdaine, B. Saphir, J. Squyres, et al. MPI-2: Extensions to the message passing interface. *University of Tennessee, available online at http://www. mpiforum. org/docs/docs. html*, 1997.

[5] H. Kamal and A. Wagner. An integrated fine-grain runtime system for MPI. *Computing*, 96(4):293–309, 2014.

[6] MPI Forum. MPI: A message-passing interface standard version 3.1. Technical report, University of Tennessee, Knoxville, 2015.

[7] A. Skjellum, A. Kanevsky, Y. S. Dandass, J. Watts, S. Paavola, D. Cottel, G. Henley, L. S. Hebert, Z. Cui, and A. Rounbehler. The real-time message passing interface standard (MPI/RT-1.1). *Concurrency and Computation: Practice and Experience*, 16(S1):Si–S322, 2004.

[8] R. Thakur and W. Gropp. Test suite for evaluating performance of mpi implementations that support mpi thread multiple. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 46–55. Springer, 2007.