LCI: Fast and generic concurrent communication engine

Hoang-Vu Dang
University of Illinois at Urbana-Champaign
Illinois, USA
hdang8@illinois.edu

Marc Snir
University of Illinois at Urbana-Champaign
Illinois, USA
snir@illinois.edu

ABSTRACT

Communication hardware and software have a significant impact on the performance of clusters and supercomputers. Message-passing model and the Message-Passing Interface (MPI) is a widely used model of communications in the High-Performance Computing (HPC) community. However, MPI has recently faced new challenges due to the emergence of many-core architecture and of programming models with dynamic task parallelism, assuming a large number of concurrent threads. These applications come from important classes of applications such as graph and data analytics.

In this thesis, we studied MPI under the new assumptions. We identified several factors in the standard which were inherently problematic for scalability and performance. Next, we analyzed graph, threading and data-flow frameworks to understand the communication. We then proposed a communication engine (LCI) targetting these frameworks. Our thesis benefits MPI by developing several patches in production MPI. Furthermore, LCI presents a simple and ground-up design which benefits various frameworks of study.

CCS CONCEPTS

- **Computer systems organization**;

ACM Reference Format:
Hoang-Vu Dang and Marc Snir. 2018. LCI: Fast and generic concurrent communication engine. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/0000001.0000001

1 BACKGROUND

Supercomputers are shifting towards increasing heterogeneity within compute nodes, with : machines different, specialized compute cores and accelerators and different types of memories. In addition, the speed of a node can increase or decrease over time due to power management, leading to to heterogeneity in time. Application codes are also becoming more adaptive – saving operations by adjusting to the evolution of the system they simulate. The complexity of such systems and their dynamic nature preclude static resource allocation and require increasing reliance on a dynamic runtime scheduler.

One programming model that is amenable to such approach is that of the *Asynchronous Task Model* (ATM): A program consists of a set of lightweight tasks that are dynamically scheduled when their dependencies have been satisfied. A runtime tracks those dependencies and schedule tasks so as to keep all compute devices busy and reduce communication. Recent examples of such models include Legion [1], PaRSEC [3], and the task model of OpenMP [15]. An ATM runtime is also the foundation for systems such as HPX [11], Charm++ [12] and Chapel [4]. The concepts however are not new: ATM is the descendant of hybrid dataflow modes that were studied in the 80's [9].

Previous research on such systems focused on shared memory parallelism and layered internode communication atop existing communication libraries such as MPI [13] or GASNet [2]. Neither was designed to support ATM: MPI was designed around the send-receive paradigm and GASNet was designed to support PGAS languages. The basic communication and synchronization paradigm of an ATM runtime is that of producer-consumer. The semantic mismatch between the functionality provided by MPI or GASNet and the needs of higher-level runtimes mean that the communication overhead at the level of MPI or GASNet is a small fraction of the total end-to-end overhead. It also means that the ability of Network Interface Controllers (NICs) to accelerate producer-consumer communication and coordination is limited as designed functionality is not exposed in MPI or GASNet. In particular, performance deteriorates with a large number of concurrent communications.

New lower-level communication libraries are emerging as newer standards that might replace the InfiniBand Verbs interface [10] such as Libfabric [8] and UCX [14]. Similarly, to past efforts, these libraries were designed to accelerate current communication patterns (message-passing, active messages), not to support new communication and synchronization patterns.

Our goal is to design and implement LCI, a communication library with new communication primitives to enable fast producer-consumer coordination with no serial bottleneck, to manage irregular, fine grain communication, to take advantage of early binding for recurring and concurrent communication patterns and to provide new efficient synchronization mechanisms. LCI takes advantages of the lessons learned from the study of existing performance bottlenecks in MPI: overheads due to MPI semantics such as wildcard matching and mutal exclusions [6, 7], overheads due to rarely used features such as derived datatypes, and memory consumption issues such as request and window management [5].

2 LCI SPECIFICATION

Endpoints. LCI communications are modeled around *endpoints*. A process may own multiple (logical) endpoints. The concept of endpoint need not be in 1-to-1 correspondence with the concept of rank in MPI. Depending on the implementation, an endpoint could

---

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/0000001.0000001

be a hardware context (thus reducing the need to multiplex different types of communication through the same hardware pipeline, and providing better performance isolation); or it could be one virtual client of many using the same hardware channel. Also, an endpoint may correspond to multiple low-level protocols (e.g., shared memory and Infiniband). A parallel application starts with one endpoint at each involved process. All these initial endpoints are connected and can be used for communication between all involved processes. New endpoints can be created locally and exchanged with other processes, in order to connect.

Producer/Consumer Specification. A basic point-to-point communication involves two processes and results in data being moved from a *source buffer* at the *producer process* to a *destination buffer* at the *consumer process*. LCI defines multiple ways to specify a buffer in a communication call: 1) *Piggy-back*: for small data that can be attached to a single packet; 2) *Explicit*: a pair of `<address, length>` is specified, which represents a contiguous buffer starting at virtual address `address` and containing `length` bytes; 3) *Dynamic allocation:* an (custom) *allocator* is specified which allocates dynamically the destination buffer. When an allocator is used, information on the allocated buffer is retrieved via the completion mechanism.

Completion events. Local completion of communication in LCI is specified through *completion events*. A completion event at the producer means that the source buffer can be reused; at the consumer it means that the destination buffer is ready for consumption. In order to reduce overheads, the completion mechanism is specified at the endpoint creation as a property of the endpoint, The following mechanisms are supported: 1) *Completion queue*: Entries providing information on completed communications are appended to a *completion queue*. The completion queue entries include the message metadata, origin endpoint and if needed, piggy back data or a buffer descriptor. 2) *Synchronizer*: A synchronization method that is applied to a synchronization object specified in the call. The synchronizer is an interface that can be overridden by the thread package so that it can be inlined. The synchronization object may have extra fields to hold the message metadata and the data itself, or a buffer descriptor. The simplest synchronizer provided by default is to set a flag. 3) *Generic Handler*: The call specifies a handler to execute upon completion. The handler is passed the message metadata and either the piggy back data or a buffer descriptor. This is similar to an Active Message.

One-sided and two-sided communication. In two-sided communication, the producer (sender) specifies the source buffer, the source endpoint where the completion event will be triggered and the destination endpoint where the data will be routed to. The consumer (receiver) specifies the destination buffer and the destination end-point to receive the data and trigger completion. Two-sided communication can specify a tag for matching between sender and receiver; however, no wildcard matching is allowed for scalable message-matching with many threads [7]. The one-sided communication is similar but takes effect by only one call, either on the producer side (Put) or the consumer side (Get); this call specifies additionally, the source/destination buffer depending on whether this is producer or consumer side.

3 PRELIMINARY RESULTS

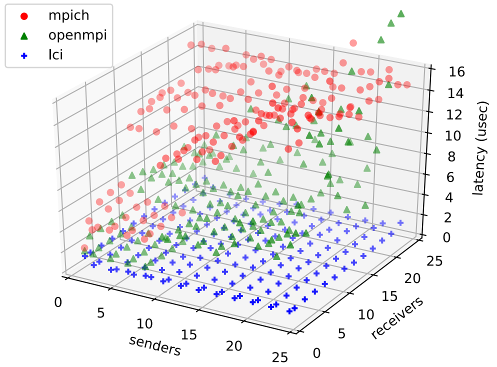

Figure 1: LCI outperforms MPICH and OpenMPI on OpenMP using multi-threaded point-to-point varying number of sender or receiver threads. Performance is done on Stampede2 cluster.

REFERENCES

[1] M. E. Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[2] D. Bonachea. GASNet specification, v1. l. *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1âĂŞ2):37 – 51, 2012.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[5] H.-V. Dang, A. Brooks, N. Dryden, M. Snir, R. Dathathri, G. Gill, L. Hoang, and K. Pingali. A lightweight communication runtime for distributed graph analytics. In *International Parallel and Distributed Processing Symposium*, IPDPS, 2018.

[6] H.-V. Dang, S. Seo, A. Amer, and P. Balaji. Advanced thread synchronization for multithreaded mpi implementations. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 314–324. IEEE Press, 2017.

[7] H.-V. Dang, M. Snir, and W. Gropp. Towards millions of communicating threads. In *European MPI group meeting (EuroMPI) 2016, Best paper*, 2016.

[8] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. A brief introduction to the OpenFabrics interfaces-a new network API for maximizing high performance application efficiency. In *HOTI*, pages 34–39. IEEE, 2015.

[9] R. A. Iannucci. *Parallel machines: parallel machine languages. The emergence of hybrid dataflow computer architectures*, volume 96. Springer Science & Business Media, 2012.

[10] InfiniBand Trade Association. Infiniband architecture specification volume 1&2, release 1.3, 2015.

[11] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[12] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[13] MPI Forum. MPI: A message-passing interface standard version 3.1. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

[14] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. UCX: an open source framework for HPC network APIs and beyond. In *HOTI*, pages 40–43. IEEE, 2015.

[15] R. van der Pas, E. Stotzer, and C. Terboven. *Using OpenMP – The Next Step*. The MIT Press, 2017.