LCI: Low-level communication interface for asynchronous distributed-memory task model

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

CCS CONCEPTS

• **Computer systems organization**;

ACM Reference Format:
Hoang-Vu Dang and Marc Snir. 2018. LCI: Low-level communication interface for asynchronous distributed-memory task model. 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 increasingly more heterogeneity within a compute node: machines increasingly include different specialized compute cores and different types of memories; the speed of a node can increase or decrease over time due to power management; algorithms might take advantage of heterogeneity in problem space and adapt during runtime. To make efficient use of such systems, inter- and intra-node parallelism must be exploited while addressing problems such as load balancing, latency, and communication management. This has placed a huge burden on the typical user/programmer to executes/writes efficient parallel applications. A remedy has been resurging is 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 [13]. An ATM runtime is also the foundation for systems such as HPX [9], Charm++ [10] and Chapel [4]. The concepts are not new: ATM is the descendant of hybrid dataflow modes that were studied in the 80's [7].

Previous research however focused on shared memory parallelism and layered internode communication atop existing communication libraries such as MPI [11] or GASNet [2]. Neither was designed to support ATMs: MPI was designed around the send-receive paradigm and GASNet was designed to support PGAS languages. The implementation of producer-consumer synchronization atop such libraries result in unnecessary overheads. In particular, performance deteriorates whith a large number of concurrent communication. Furthermore, semantic mismatches between the functionality provided by current NICs 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. New lower-level communication libraries are emerging as newer standards that might replace the InfiniBand Verbs interface [8] such as Libfabric [6] and UCX [12]. 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 goals are to design and implement LCI, a communication library with new communication primitives to enable fast coordination with no serial bottleneck, to manage irregular, fine grain communication, to take advantage of early binding for recurring communication patterns and to provide new efficient synchronization mechanisms. The communication library will be tightly integrated with task schedulers and memory managers and work with multiple CPUs/GPUs. LCI takes advantages from the studies of MPI semantics and its existing performance issues with high-thread counts, overheads from unused features such as datatypes, wildcard tag-matching, implicit request and memory consumption [5].

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

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

2 LCI SPECIFICATION

2.1 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 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.

2.2 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 many ways to specify a buffer in a communication call. It can be *explicit* where a pair of `<address, length>` is specified, which represents a contiguous buffer starting at virtual address `address` and containing `length` bytes; or *dynamic allocation:* an *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.

2.3 Completion events

Completion in LCI is specified through *completion events*. At the source endpoint, a completion is *local*, i.e. the source buffer can be reused; while at destination endpoint, the completion event is triggered when the data has fully arrived. Since multiplexing data can be expensive, completion mechanism is specified at the endpoint creation as the property of the endpoint, as one of the following types.

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.

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 inline. The synchronization object may have extra fields to hold the message metadata and the data itself, or a buffer descriptor. The most simple synchronizer provided by default is to set a flag.

Generic Handler: The call specifies a handler to execute upon completion. The handler is passed message metadata and either the piggy back data or a buffer descriptor. This is similar to an Active Message.

2.4 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. 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 [5]. 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 specification depending on whether this is producer or consumer side.

3 PREMILINARY RESULTS

We integrate LCI with our customized thread scheduler and compare its performance with traditional approach using MPI+OpenMP (latest MPICH and OpenMPI). This micro-benchmark spawns two processes in two different nodes, each creates a number of threads. A thread in the sender process issues a send then a receive, while a thread in the receiver process issue a receive then a send that match a single thread of the sender process using appropriate tag. The performance shows LCI consistently maintains low latency per message pair, while MPI+OpenMP degrades with more send/receive threads.

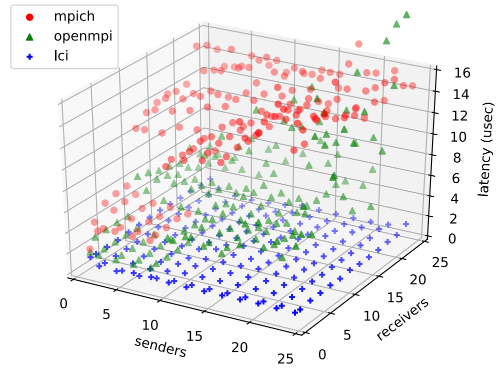

Figure 1: LCI outperforms MPICH and OpenMPI on OpenMP using multi-threaded point-to-point varying number of sender or receiver threads. Performance is done on Stampede2 cluster.

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, M. Snir, and W. Gropp. Towards millions of communicating threads. In *European MPI group meeting (EuroMPI) 2016, Best paper*, 2016.

[6] 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 *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 34–39. IEEE, 2015.

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

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

[9] 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.

[10] 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.

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

[12] 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 *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 40–43. IEEE, 2015.

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