Anycast: Rootless Broadcasting with MPI

Tonglin Li
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, California
tonglinli@lbl.gov

Quincey Koziol
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, California
koziol@lbl.gov

Houjun Tang
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
htang4@lbl.gov

Suren Byna
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
sbyna@lbl.gov

ABSTRACT

Flexible communication routines that enable high-performance computing (HPC) applications to operate with less synchronization are a great benefit to application algorithm creation and developer productivity. Further, communication operations that reduce synchronization in HPC applications while continuing to scale well are critical to application performance in the exascale era. We present an algorithm that allows an application to scalably broadcast a message from any MPI rank without specifying the origin of the message in advance, demonstrating application development flexibility, highly scalable operation, and reduced application synchronization.

CCS CONCEPTS

• **Software system structures → Software system models**; **Ultra-large-scale systems**;

KEYWORDS

Broadcast, MPI, High performance computing

ACM Reference Format:
Tonglin Li, Quincey Koziol, Houjun Tang, and Suren Byna. 2018. Anycast: Rootless Broadcasting with MPI. In *Proceedings of ExaMPI 2018.* ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over the past two decades, high-performance computing (HPC) systems' performance gains have been increasingly reliant on higher levels of parallelism. To use these highly parallel computing resources efficiently, the majority of scientific applications are written using the MPI (Message Passing Interface) programming framework, where computation is performed by multiple processes that communicate and coordinate using MPI capabilities. During that time, MPI has evolved into the most mature and reliable communication protocol for parallel programming. Other parallel programming paradigms such as Map-Reduce have emerged and achieved success in cloud computing environments, but MPI is the dominant force in HPC environments.

Despite MPI's extensive functionality, it still has limitations. For example, many collective operations, such as `MPI_Bcast` and `MPI_Iallreduce`, require a "root" process to be specified. This requirement assumes that an application developer always knows which rank will initiate a broadcast operation when writing the program.

Imagine an application where the results of each process's compute operation need to be shared with all the other processes in order to update a shared data structure, but each process can otherwise proceed independently from the others. For example, in PDC [5, 6], an object-centric data management system, the PDC servers and clients need a similar rootless broadcast for broadcasting notifications about metadata operations that change the state of an object. In the current implementation of MPI, each process with results must report to a predefined dedicated process (the coordinator), and then the coordinator must broadcast the information to the other processes. Clearly, when such an application scales to many MPI processes, the coordinator will become a severe bottleneck.

To address this problem, we have built a high performance "rootless" broadcast mechanism, called *Anycast*, which allows any rank to initiate broadcasts at any time. This mechanism uses MPI message forwarding over a skip-ring overlay topology to achieve logarithmic time message delivery. We have tested the algorithm on the Cori system [2] at NERSC, and the preliminary results show very good performance and excellent scalability.

2 DESIGN AND IMPLEMENTATION

2.1 Skip list and skip ring

A skip list is an ordered data structure used in some data store systems, such as Redis [3] and SkipDB [4], due to its fast search and updates. The search and update time complexity is *O(log(n))* on average, and *O(n)* in the worst case, where *n* is the number of data nodes. Figure 1 shows a skip list with 8 data nodes.

In our design, we modify the structure of a skip list and make the final data nodes at each level of the list (nodes 4, 6 and 7 in figure 2) point to the head node (0 in in figure 2) to form a multi-level ring,

ExaMPI 2018, November 2018, Dallas, Texas USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

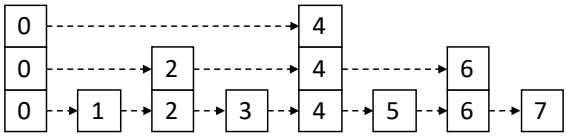

Figure 1: A skip list with 8 nodes.

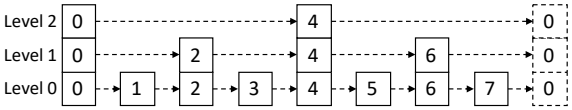

Figure 2: A skip ring with 8 nodes. Nodes 0 and 4 have the highest level and most outbound channels.

and we call it a *skip ring*. Then we build an overlay network on an MPI communicator with a skip ring topology. In this overlay network, each skip ring node represents an MPI rank, and each arrow represents a point-to-point communication channel. All message passing in an anycast happens only on these channels. Once MPI ranks are organized in this way, each rank only has a small list of channels used for sending. Receiving channels don't need to be specified, because it is implied by the successor rank's sending channels. Inbound messages are captured by posting an `MPI_Irecv()` with the `MPI_ANY_SOURCE` tag on each receiver, which can then be queried for the message size and sender rank.

2.2 Forwarding algorithm

When an MPI rank wants to initiate an anycast, it only needs to send the message through its outbound channels. Every rank periodically polls the status of its posted `MPI_Irecv()` operation and when a rank receives a message, it forwards the message through its outbound channels. Since the outbound channels for each rank are predefined at the algorithm's initialization, the forwarding routine for each rank of the ring behaves identically, simplifying the algorithm's implementation. To prevent endless message forwarding and duplicated messages around the ring, we set two additional rules for forwarding messages:

1) Message forwarding only happens from nodes with a higher level to ones with a lower level, unless the message is received from a node with a lower skip ring level. A message received from a node with a lower skip ring level than the receiving node is a newly broadcast message and must be forwarded to all outbound channels of the receiving node.

2) A message will not be forwarded if the destination node is located **after** the origin of that message on the ring. This ensures that forwarded message don't pass their origin and create duplicated messages.

As shown in the example in figure 3, this algorithm ensures that each message will be sent using only **n** messages, and received by all nodes in *O(log(n))* time, on average.

2.3 Source code

Anycast code is published under a BSD license on Bitbucket and available at [1].

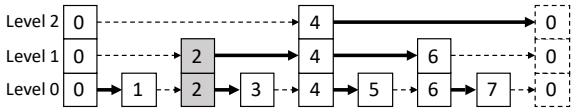

Figure 3: Anycast message routes. Rank 2 initiates anycast. Note that rank 2 can send to rank 4 (higher level), and rank 4 can send to rank 0 (same level) because this is a new message.

3 PRELIMINARY RESULTS

3.1 Experiment setup

We conducted preliminary benchmarking for anycast on Cori, NERSC's latest system. The scale tested is from 16 to 512 MPI ranks. After seeing significant performance loss with hyperthreading enabled, we assigned each MPI process to one physical core, thus using up to 512 hardware cores. Message sizes range from 10 bytes to 10,000 bytes. For each round of experiment, we execute 10,000 anycasts in a tight loop on rank 0, and measure the time on all other ranks. Since messages in anycast don't arrive their destination at the same time, we consider the longest running time among all ranks to be the anycast finishing time. We ran at least 4 rounds of experiments and calculated the average anycast latency for each of the scale / message size combinations.

3.2 Benchmark results

In figure 4 we can see that the measured latency is not sensitive to message size, implying that the system bandwidth is far from being saturated and much larger message sizes should make more visible differences. Despite the large variance in the measured values, the trends only show a slight increase, indicating an excellent scalability (although the ideal scalability curve should be flat).

We tested anycast over a wide variety of scales and messages sizes. Figure 4 shows a somewhat irregular change of latency correlate with scales, this is likely due to the network jitter from testing on a production system. Although the average latencies are calculated with a large amount of requests (40K per experiment) the system noise is still not smoothed out, especially in the line associated with 100 bytes message size. Although we have not yet explored a larger parameter space to figure out the reason, we believe the current variation is due to executing our experiments within a small time window and on the same computing resource allocation, causing the coarse granularity as system noise impact became noticeable.

4 FUTURE WORK AND CONCLUSION

Although the current implementation shows excellent scalability and great flexibility for application developers, there are two drawbacks to the current implementation. First, the skip ring must be initialized on an MPI communicator, and the resulting "rootless handle" passed to the anycast broadcast and receive operations. Secondly, because the application doesn't know when another rank will send a rootless broadcast message, the rootless receive operation must be polled periodically by the application. Both of these

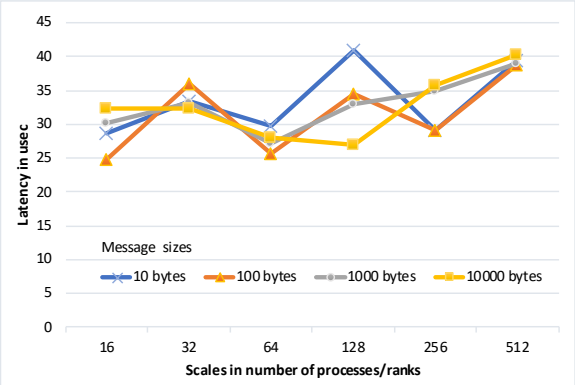

Figure 4: Benchmark results show excellent scalability with various message sizes. Note that the latencies with different message sizes are very close to each other.

restrictions could be addressed by moving the anycast implementation into the MPI library properly, making them even easier for application developers to use.

Based on the experience of building anycast and our development needs, we have started developing more rootless versions of MPI operations, starting with a rootless allreduce. If a robust set of rootless operations are able to be created, we will propose them as an addition to the MPI standard.

With the help of the flexibility brought by rootless operations, MPI application developers can program with simpler logic structures and still enjoy scalable, less synchronized communication.

ACKNOWLEDGMENTS

The work used computing resource at National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory (LBL), and is supported by the DOE funding No.XXXXX.

REFERENCES

[1] Anycast git repo. https://tonylbl@bitbucket.org/tonylbl/shared_ledger_rma.git. Accessed: 2018-10-9.
[2] Cori. http://www.nersc.gov/users/computational-systems/cori. Accessed: 2018-10-9.
[3] Redis. https://en.wikipedia.org/wiki/Redis. Accessed: 2018-10-9.
[4] Skipdb. http://dekorte.com/projects/opensource/skipdb/. Accessed: 2018-10-9.
[5] Tang, H., Byna, S., Dong, B., Liu, J., and Koziol, Q. SoMeta: Scalable Object-Centric Metadata Management for High Performance Computing. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept 2017), pp. 359–369.
[6] Tang, H., Byna, S., Tessier, F., Wang, T., Dong, B., Mu, J., Koziol, Q., Soumagne, J., Vishwanath, V., Liu, J., and Warren, R. Toward Scalable and Asynchronous Object-centric Data Management for HPC. In *CCGRID* (2018).