Optimizing Charm++ over MPI

Ralf Gunter, David Goodell, James Dinan, and Pavan Balaji

Mathematics and Computer Science Division
Argonne National Laboratory
{rgunter,goodell,dinan,balaji}@mcs.anl.gov

March 25, 2013

Abstract

Charm++ may employ any of a myriad network-specific APIs for handling communication, which are usually promoted as being faster than its catch-all MPI module. Such a performance difference not only causes development effort to be spent on tuning vendor-specific APIs but also discourages hybrid Charm++/MPI applications. We investigate this disparity across several machines and applications, ranging from small InfiniBand clusters to Blue Gene/Q supercomputers and from synthetic benchmarks to large-scale biochemistry codes. We demonstrate the use of one feature from the recent MPI-3 standard to bridge this gap where applicable, and we discuss what can be done today.

1 Introduction

Our work identifies inefficiencies in both design and implementation that impair the usage of MPI [4] as a Charm++ [2] networking substrate. Incongruences between the two programming models imply a likely loss of performance, irrespective of platform-specific details. We also propose concrete interoperability opportunities that substantiate the optimizations presented.

A number of potential technical culprits were investigated, from unexpected message queue lengths to the frequency of calls to the MPI progress engine. We investigated the effects of the alternating computation versus communication phases of single-threaded Charm++. We also examined the trade-offs of utilizing eager versus rendezvous protocols in MPI. The most notable performance improvement came from supporting communicator-specific rendezvous thresholds, a feature made possible by the recent MPI-3 standard.

Regardless of specific implementation details, we believe there is a fundamental difference between the expected-message nature of MPI and the unexpected-message behavior of Charm++. This difference makes the former less suited to performing the communication responsibilities of the latter.

We motivate our changes with benchmarks (both artificial and scientific) that make novel use of the hitherto inefficient Charm++/MPI interoperability interface.

The MPI network module

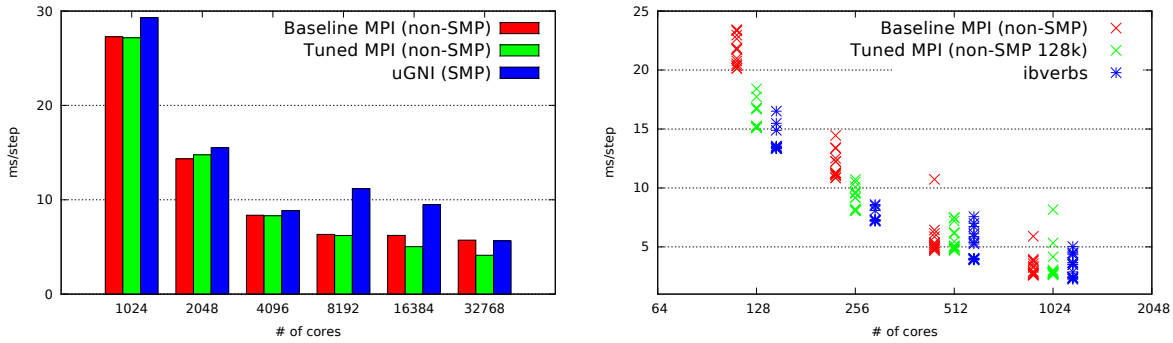
Many *chares* map to the same MPI rank, a fact that will be important later.

The following is the default algorithm for message transfers.

Sending: Messages are sent asynchronously by using MPI_Isend with a fixed tag, and the request is enqueued. From time to time, the RTS uses MPI_Test to determine (and delete) those messages that have completed.

Receiving: Whenever control is relinquished to its scheduler, Charm++ goes into a polling loop comprising an MPI_Iprobe followed (if it succeeds) by an MPI_Recv matching `MPI_ANY_SRC` and `MPI_ANY_TAG`. All messages share the same tag and hence are not prioritized in any way within the module.

Optional changes (disabled by default) include using a rendezvous-like control message scheme, changing the MPI_Recv into an MPI_Irecv call, and having a set of preposted MPI_Irecv to avoid unexpected messages. There is also an SMP build where a user-configurable set of threads, parked in the algorithm described above, is dedicated to handling communication.



(a) Blue Waters (Cray Gemini)

(b) Fusion (InfiniBand)

Figure 1: Comparison of NAMD over MPI vs the native Charm++ module

2 Analysis of the default MPI module

Several mismatches between the Charm++ programming model and the way MPI is designed to work were investigated in order to explain the performance differences seen above, including the following.

Expected vs unexpected messages: Because of the asynchronous model, messages in Charm++ are always unexpected. As a result, typically an extra `memcpy` is done between the temporary buffer, where the MPI implementation stores the message, and the final user buffer (managed by Charm++). In practice, enabling the preposted MPI_Irecv above does not improve performance; we suspect that the costs involved in asynchronous MPI calls counterbalance any benefit gained by avoiding the `memcpy`.

A related concern in pure MPI applications is the length of the unexpected message queue, where matching a given sender/tag pair might involve numerous passes through this queue. In our context, however, the use of `MPI_ANY_TAG` and `MPI_ANY_SRC` will match the first enqueued unexpected message, and hence no traversal is necessary.

Prioritization: Unlike native APIs, MPI has no support for prioritized messages.

Oversubscription: Charm++ encourages oversubscription of processing elements, which helps ease work starvation as well as load imbalance. Hence, in non-SMP mode, all chares mapped to an MPI rank will momentarily block while handling communication. This situation is especially aggravating in the case of longer messages, which use the MPI rendezvous protocol.

Sender/receiver synchronization: Message transfer is initialized only when both endpoints are in the MPI progress engine. Therefore, it is beneficial to use a messaging protocol with as little synchronization as possible, hence the eager threshold change in the next section.

3 The eager/rendezvous threshold

We found that changing the default MPI eager threshold leads to a faster initialization phase in NAMD, as well as an improvement in timestep duration, as seen in Figure 1a. By removing two extra synchronization steps, the eager protocol more closely resembles the asynchrony of the Charm++ model.

Per-communicator thresholds

Resetting the eager threshold via an environment variable has the downside of affecting subsequent MPI programs, whose ideal value might vary. Although sometimes this can be mitigated by adjusting the thresholds as we switch from Charm++ to MPI codes, we decided to take the more generic route of making communicators use different values. This feature, made possible by MPI-3 [4] and added in MPICH [5], is especially useful in hybrid Charm++/MPI applications, where there might be concurrent communication from both runtimes.

4 Related work

While we have investigated the implementation of Charm++ over MPI, Huang et al. explored the converse in the context of AMPI [1]. This model virtualizes MPI processes over Charm++ processing elements.

More recently, Sun et al. [6] examined Charm++ implementation issues on current Cray Gemini-based systems while Kumar et al. [3] studied IBM Blue Gene/Q systems. These works look at optimizations to the native (non-MPI) networking layer built into Charm++. The techniques may serve as the basis for future Charm++-over-MPI optimizations to help mitigate the programming model impedance mismatch.

5 Conclusion

MPI and Charm++ cater to different programming paradigms: the former favors BSP-styled codes, whereas the latter supports a more asynchronous design. Therefore, while the former may be used as the networking engine for the latter, this comes at a small but perceptible cost. After investigating many MPI technicalities as possible causes of this gap in practice, we noticed that tweaking its eager threshold leads to better performance for NAMD on Cray machines. We then implemented in MPICH a way of setting these thresholds at the communicator granularity, so as to provide applications with finer tuning mechanisms, such as in hybrid Charm++/MPI codes.

Acknowledgments

This work was supported by the U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, TX, October 2003.

[2] L.V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[3] Sameer Kumar, Yanhua Sun, and Laxmikant V. Kalé. Acceleration of an asynchronous message driven programming paradigm on IBM Blue Gene/Q. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, MA, 2013. IEEE Computer Society. (to appear).

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, September 2012. `http://www.mpi-forum.org/docs/docs.html`.

[5] MPICH. `http://www.mpich.org/`.

[6] Yanhua Sun, Gengbin Zheng, Chao Mei, Eric J. Bohm, Terry Jones, Laxmikant V. Kalé, and James C. Phillips. Optimizing fine-grained communication in a biomolecular simulation application on Cray XK6. In *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, Salt Lake City, UT, November 2012.

Government license