Techniques for Enabling Highly Efficient Message Passing on Many-Core Architectures

Min Si
University of Tokyo
Tokyo, Japan
msi@il.is.s.u-tokyo.ac.jp

Pavan Balaji (Co-advisor)
Argonne National Laboratory
Argonne, IL, USA
balaji@mcs.anl.gov

Yutaka Ishikawa (Advisor)
RIKEN Advanced Institute for Computational Science
Kobe, JAPAN
yutaka.ishikawa@riken.jp

***Abstract*—Many-core architecture provides a massively parallel environment with dozens of cores and hundreds of hardware threads. Scientific application programmers are increasingly looking at ways to utilize such large numbers of lightweight cores for various programming models. Efficiently executing these models on massively parallel many-core environments is not easy, however and performance may be degraded in various ways. The first author's doctoral research focuses on exploiting the capabilities of many-core architectures on widely used MPI implementations. While application programmers have studied several approaches to achieve better parallelism and resource sharing, many of those approaches still face communication problems that degrade performance. In the thesis, we investigate the characteristics of MPI on such massively threaded architectures and propose two efficient strategies—a multithreaded MPI approach and a process-based asynchronous model—to optimize MPI communication for modern scientific applications.**

***Keywords*-many-core; MPI; hybrid programming model; one-sided; asynchronous progress**

I. Introduction

Since multicore processors have become the most common processor architectures today, the only way to improve the performance for high-end processors is to add more threads and cores. Many-core architectures, such as Intel Xeon Phi and Blue Gene/Q, provide such a massively parallel environment with dozens of cores and hundreds of hardware threads. More and more scientific application programmers have begun investigating ways to utilize such architecture for scaling application performance.

Although many-core architecture can provide the enormous power of parallel computing, application performance may still be restricted in various ways. Two characteristics of such hardware have to be taken into account. First, cores are designed to be simple and low frequency for a better performance-to-energy ratio; thus, execution on a single core could result in extreme performance degradation. Second, the number of cores is increasing at a faster rate than other on-chip resources (e.g., memory), potentially resulting in scalability bottlenecks.

To better utilize such hardware resources, application programmers have studied several approaches that provide better parallelism and resource sharing for different scientific applications. Many of those approaches, however, still face communication problems that may result in performance degradation. This doctoral research aims to exploit the capabilities of many-core architectures on the widely used message-passing model and propose techniques for solving existing communication problems. In this thesis, we focus on optimizing the communication of the two most popular programming models used in modern applications: a hybrid MPI+threads model and an MPI one-sided communication model.

Communication optimization in hybrid MPI+threads model. An increasing number of applications are looking at hybrid programming models, frequently called "MPI+threads," to allow resources to be shared between different cores on the node. A common mode of operation in such hybrid models involves using multiple threads to parallelize computation within the node, but using only one thread to issue MPI communication. Although such a mode achieves significant improvement in floating-point computing by massive parallelism, it also means that most of the threads are idle during MPI calls, a situation that translates to underutilized hardware cores. Furthermore, since MPI communication performs only on a single lightweight core, this mode may even result in performance degradation.

To resolve the problems in the MPI communication of hybrid model, we present MT-MPI [1], an internally multithreaded MPI that transparently coordinates with the threading runtime system to share idle threads with the application in order to parallelize MPI internal processing such as derived datatype communication, shared-memory communication, and network I/O operations.

Optimization for MPI one-sided communication. For applications with large memory requirements, developers start sharing memory resources across nodes through a global shared address space implemented by employing MPI one-sided communication [2]. The MPI-2 and MPI-3 standards [3] introduced one-sided communication (also known as remote memory access or RMA), which allows one process to specify all communication parameters for both sender and receiver. Thus a process can access memory regions of other processes in the system without the target process explicitly needing to receive or process the message. Although such communication semantics can asynchronously handle communication progress and hence hide communication overheads from computation, it is not truly asynchronous in most MPI implementations. For example, although contiguous PUT/GET MPI RMA communica-

```
#pragma omp parallel
{

  /* user computation */

}

MPI_Function();
```

(a) Outside a parallel region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp single
  {
    MPI_Function();
  }
}
```

(b) Inside omp single region

Figure 1. Example of different use cases in hybrid MPI+OpenMP.

tion can be implemented in hardware on RDMA-supported networks such as InfiniBand, thus allowing the hardware to asynchronously handle its progress semantics, complex RMA communication such as an accumulate operation on a 3D subarray must still be done in software within the MPI implementation. Consequently, the operation cannot complete at the target process without explicitly making MPI progress and thus may cause arbitrarily long delays if the target process is busy computing outside the MPI stack.

To resolve the problem of asynchronous progress, we propose Casper [4], a process-based asynchronous progress model for MPI one-sided communication on multicore and many-core architectures, that keeps aside a small user-specified number of cores as background "ghost processes" to help asynchronous progress. The philosophy of Casper is centered on the notion that since the number of available cores in modern many-core systems is increasing rapidly, some of the cores might not always be busy with computation and can be dedicated to helping with asynchronous progress.

In summary, this Ph.D. thesis aims to enable highly efficient message passing on many-core architectures for various kinds of scientific applications. Two techniques are proposed to address different communication issues existing in modern applications. In Sections II and III, we separately present their current state and sketch the next steps.

II. MULTITHREADED MPI

In this section, we present our first technique that improves communication for the hybrid MPI+threads programming models.

In the common mode of hybrid MPI+OpenMP applications, multiple threads are used to parallelize the computation, while one of the threads handles MPI communication (i.e., MPI `FUNNELED` or `SERIALIZED` thread-safety mode). This is achieved, for example, by placing MPI calls in OpenMP single sections (Figure 1(b)) or outside the OpenMP parallel regions (Figure 1(a)). However, such a model often means that the OpenMP threads are active only in the computation phase and idle during MPI calls, resulting in wasted computational resources. Moreover, since only a single lightweight core issues MPI communication, performance may even be degraded.

In [1], we focused on optimizing the FUNNELED/SERIALIZED modes. We presented MT-MPI, an internally multithreaded MPI implementation that transparently coordinates with the threading runtime system to share idle threads with the application. We designed MT-MPI in the context of OpenMP, which serves as a common threading runtime system for the application and MPI. MT-MPI employs application idle threads to parallelize MPI communication and increases resource utilization.

Specifically, we modified the MPI implementation to parallelize its internal processing using a potentially nested OpenMP parallel instantiation (i.e., one nested OpenMP parallel block inside the MPI call in Figure 1(b), which is inside another parallel block). We expected that such a model would allow both the application and the MPI implementation to expose their parallelism requirements to the OpenMP runtime, which in turn can schedule them on the available computational resources. In practice, however, this model has multiple challenges. One challenge is that the "parallelism-friendly" algorithms utilized in MPI internal processing tasks are in some cases not as efficient for sequential processing, even resulting in performance degradation with insufficient available threads. Moreover, many implementations of the OpenMP runtime do not schedule work units from nested OpenMP parallel regions efficiently. Instead, they simply create new pthreads for each nested parallel block and rely on the operating system to schedule them on the available cores, resulting in core oversubscription and performance degradation.

A. OpenMP Runtime Extension

To address these challenges, we modified the OpenMP runtime to expose information about the idle threads to the MPI implementation. We first track the number of threads that are being used by the application and the number of idle threads (e.g., the threads are waiting in an OpenMP barrier or outside an OpenMP parallel region) in the OpenMP runtime system. Then, we define a new OpenMP runtime function in order to expose such information. The expectation with this model is that MPI could query for the number of idle threads and use this information to (1) choose different algorithms that trade off between parallelism and sequential execution in order to achieve the best performance in all cases and (2) use only as many threads in the nested OpenMP region as there are idle cores, by explicitly guiding the number of threads in OpenMP (using the `num_threads` clause in OpenMP).

B. MPI Internal Parallelism

Using the information of idle threads exposed by our extended OpenMP runtime, the MPI implementation can schedule its internal parallelism efficiently to obtain performance improvements. We demonstrate the benefit of such internal parallelism for three aspects of the MPI processing.

1) Derived Datatype Processing: Applications define derived datatypes to describe noncontiguous regions of memory in packing/unpacking processing or directly involved in

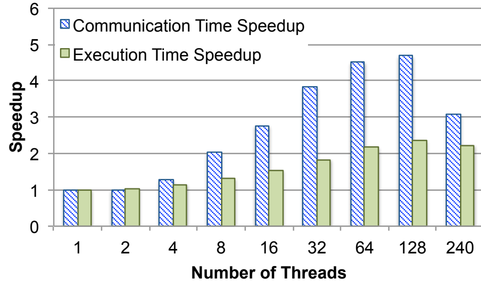


Figure 2. Hybrid NAS MG class E using 64 MPI processes.

communication that internally processes packing/unpacking. Such packing and unpacking processing stages are implemented as a set of local memory copies with no dependencies. We modified the MPI implementation to parallelize them using OpenMP.

2) Shared-Memory Communication: Most MPI implementations use a pipelined double-copy strategy [5] to transfer data between processes existing on the same node. We parallelized such copy processing by reserving multiple contiguous available chunks and concurrently coping data by utilizing multiple threads.

3) Optimizations for the InfiniBand Network: InfiniBand-supported MPI implementations always manage separate queue pairs (QPs) for communication between each pair of processes. We parallelized the MPI internal sending processing of messages issued through different QPs.

C. Evaluation

Our experimental evaluation, performed on the Stampede supercomputer at the Texas Advanced Computing Center (https://www.tacc.utexas.edu/stampede/), demonstrates the performance benefits of MT-MPI on various aspects of MPI processing. All our experiments are executed on the Xeon Phi coprocessor, with every MPI process running on a separate coprocessor. Because of space limitation, we present only one kernel benchmark in this paper.

We employed a hybrid MPI+OpenMP version of the NAS Multigrid (MG) benchmark [6]. The V-cycle multigrid algorithm performs multiple 3D halo exchanges with various dimension sizes. This approach is expected to significantly benefit from parallelized derived datatype processing because packing and unpacking are always the heaviest parts of each halo exchange communication. Figure 2 presents the speedup achieved by MT-MPI compared with the original MPICH in class E when employing 64 processes. MT-MPI helps improve the communication of MG by 4.7-fold, and the overall execution time by 2.2-fold.

D. Next Steps

To understand the behavior of MT-MPI in real-world applications, we plan to study several scientific applications that are expected to benefit from MT-MPI, such as CCSM [7], which relies on derived datatypes for communication, and PCCM2 [8], which performs large shared-memory communication.

III. Process-Based Asynchronous Progress Model

The second technique proposed in this thesis focuses on optimizing the MPI one-sided communication, which is often used to implement the PGAS model for many large-scale data-intensive applications.

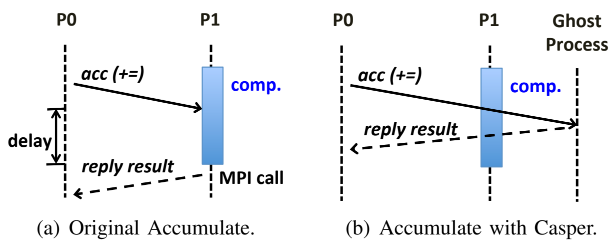
Although one-sided communication semantics can asynchronously handle communication progress, the MPI standard does not guarantee it to be truly asynchronous. In most network interfaces, complex one-sided communication operations such as noncontiguous accumulate are not natively supported. MPI implementations still require the target process to make MPI calls in order to ensure completion of such operations (Figure 3(a)).

Traditional implementations to ensure asynchronous completion of operations have relied on thread-based [9] or interrupt-based [10][11, Chapter 7] models. Each of these models has several drawbacks, however, such as the inefficient core deployment in the thread model and the expensive overheads caused by multithreading safety in the thread model and by frequent per-message interrupts in the interrupt model.

To address these drawbacks, we present Casper [4], a process-based asynchronous progress model for MPI RMA communication on multicore and many-core architectures. The central idea of Casper is to keep aside a small, user-specified number of cores on a multicore or many-core environment as “ghost processes,” which are dedicated to help asynchronous progress for user processes through appropriate memory mapping from those user processes. Unlike traditional approaches, the use of processes allows Casper to avoid expensive overheads associated with multithreaded safety or system interrupts, as well as to control the number of cores being utilized for asynchronous progress.

A. Design Overview

Casper relies on the ability of processes to expose their window memory regions by using the MPI-3 shared-memory windows interface. When the application process tries to allocate a remotely accessible memory window, Casper intercepts the call and maps such memory into the ghost processes’ address space. Casper then intercepts all RMA synchronization calls and communication operations issued to the user processes on this window and redirects them to the ghost processes instead, as shown in Figure 3(b). Since Casper does not migrate or copy the user memory regions but just maps them into the ghost processes’ address space, RMA operations that are handled in hardware see no difference in the way they behave. Operations that require remote software progress can be executed asynchronously in



(a) Original Accumulate. (b) Accumulate with Casper.

Figure 3. RMA asynchronous progress.

the ghost processes' MPI stack on the additional cores kept aside by Casper, without requiring any intervention from the application processes.

Although the core concept of this work is straightforward, the design and implementation of such a framework must take several aspects into consideration in order to ensure that correctness is maintained as required by the MPI-3 semantics. The wide variety of RMA communication and synchronization models provided by MPI could substantially complicate this work. The Casper architecture hides all this complexity from the user and manages it internally within its runtime system.

B. Evaluation

We demonstrate the benefits of Casper on the NERSC Edison Cray XC30 supercomputer (https://www.nersc.gov/users/computational-systems/edison/configuration/) with a variety of microbenchmarks and the NWChem quantum chemistry application [12]. NWChem uses the Global Arrays [13] toolkit for data movement, which has been implemented on a portable implementation over MPI RMA [2]. We present one experiment of the NWChem evaluation in this paper.

Our experiment measured the most important CCSD(T) method. The (T) portion of CCSD(T) performs a get-computation-reduce pattern, significantly benefiting from asynchronous progress. We evaluated Casper by comparing it with the original MPI and two thread-based asynchronous progress approaches. We used the same total number of cores in all approaches, some of which are dedicated to asynchronous ghost processes/threads. Figure 4 measured the C_{20} molecule, obtained from the NWChem QA test suite (`QA/tests/tce_c20_triplet`), with increasing number of cores. Casper is almost twice as fast. However, the thread-based approaches are far less effective because of the extreme performance degradation in computation from either core oversubscription or appropriation of half of the computing cores.

C. Next Steps

We plan to expand the evaluation of NWChem by analyzing more scientific models and molecule types. Specifically, we plan to focus on the self-consistent field module (SCF) which is another essential functionality of NWChem, and the polyacenes molecule.

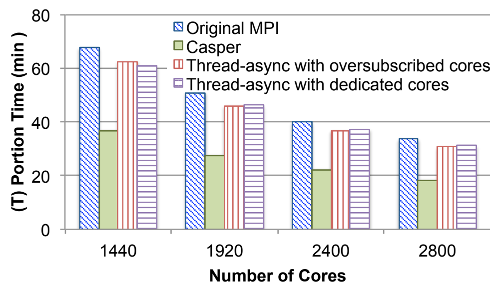


Figure 4. (T) portion of CCSD(T) for C_{20} with pVTZ.

Acknowledgments

The experimental resources for this research were provided by the Texas Advanced Computing Center (TACC) on the Stampede supercomputer and by the National Energy Research Scientific Computing Center (NERSC) on the Edison Cray XC30 supercomputer. This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

References

[1] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: Multithreaded MPI for Many-Core Environments," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 125–134.

[2] J. S. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays PGAS model using MPI one-sided communication," in *IPDPS*, May 2012.

[3] "MPI: A Message-Passing Interface Standard," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sep. 2012.

[4] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures," in *Parallel and Distributed Processing, 2015. IPDPS 2015.*

[5] D. Buntinas and G. Mercier, "Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem," in *Euro PVM/MPI*, 2006.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, 1991.

[7] R. Jacob, J. Larson, and E. Ong, "M× n communication and parallel interpolation in community climate system model version 3 using the model coupling toolkit," *International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 293–307, 2005.

[8] J. Drake, I. Foster, J. Michalakes, B. Toonen, and P. Worley, "Design and performance of a scalable parallel community climate model," *Parallel Computing*, vol. 21, no. 10, pp. 1571–1591, 1995.

[9] W. Jiang, J. Liu, H.-W. Jin, D. Panda, D. Buntinas, R. Thakur, and W. Gropp, "Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters," in *Euro PVM/MPI*, ser. Lecture Notes in Computer Science, 2004, vol. 3241, pp. 68–76.

[10] Cray Inc., "Cray Message Passing Toolkit," http://docs.cray.com/books/S-3689-24, Cray Inc., Tech. Rep., 2004.

[11] M. Gilge, *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM, Jun. 2013.

[12] E. J. Bylaska et. al., "NWChem, A Computational Chemistry Package for Parallel Computers, Version 6.3," 2013.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers," in *ACM/IEEE conference on Supercomputing*, 1994.