FAST AND GENERIC CONCURRENT MESSAGE-PASSING

BY

HOANG-VU DANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

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 with great success. However, it 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, light-weight threads. These applications come from important classes of applications such as graph and data analytics. Using MPI with these languages/runtimes is inefficient because MPI implementation is not able to perform well with threads. Using MPI as a communication middleware is also not efficient since MPI has to provide many abstractions that are not needed for many of the frameworks, thus having extra overheads.

In this thesis, we studied MPI performance under the new assumptions. We identified several factors in the message-passing model which were inherently problematic for scalability and performance. Next, we analyzed the communication of a number of graph, threading and data-flow frameworks to identify generic patterns. We then proposed a low-level communication interface (LCI) to bridge the gap between communication architecture and runtime. The core of our idea is to attach to each message a few simple operations which fit better with the current hardware and can be implemented efficiently. We show that with only a few carefully chosen primitives and appropriate design, message-passing under this interface can easily outperform production MPI when running atop of multi-threaded environment. Further, using LCI is simple for various types of usage.

To my parents, my sister, my dear wife and child for their love and support.

ACKNOWLEDGMENTS

Foremost, it is my advisor, Prof. Marc Snir, who brought me to University of Illinois that made all of this work possible. Marc was the ultimate advisor for my work in communication runtime for an obvious reason. I thought it was my luck that Marc picked me as his disciple back then, and this is still true. One thing that I love working with Marc is his ability to convert my fuzzy explanation of technical problems into concrete mathematical formulation. It shows how great of a mind he has and how great of a listener he is. Marc had given me a great environment to grow as an independent researcher, and with great ideas whenever I got stuck. More importantly, Marc is more than an advisor, he is a friend where I can share my thought freely both about work, and about family.

The work presented in this dissertation have built from results of several researchers, with the helps and suggestions from many others who have reviewed and commented on our published papers. This dissertation, clearly cannot be possible without feedbacks of the committee members. Prof. William Gropp, who has introduced me to the MPICH implementations and its problems, and given me tremendous feedback from beginning to end. Prof. Sarita Adve and Prof. Torsten Hoefler, who are kind enough to review my dissertation as well as giving me chances to work on my writing and improve my results.

I met Dr. Sangmin Seo at Argonne National Lab, who was my great mentor and also a great friend. Sangmin is a Postdoctoral scholar and researcher that I wish I will become. He is highly technical, highly critical, and at the same time calm and humble. I only spent three months with him, but I have learned much from him both from the technical side, and the organizational side. Sangmin helped me in the MPI project, and was really the person who made that a successful three-month project. If Sangmin had not returned to Korea, I definitely seek a chance to work with him again.

Prof. Bertil Schmidt at the University of Mainz was my first advisor after I obtained my Bachelor degree, who introduced me to the field of parallel computing. He brought me to Germany and helped me apply for a good graduate program. Because of Bertil, I was very well-prepared for several difficult classes at UIUC, especially for dealing with GPUs, where we wrote a book chapter together. Most importantly, he helped me begin my academic career, which I never thought of before.

Without friends and lab buddies, my graduate student life could have been so dull, especially during the bitter winter of Champaign-Urbana. Alex Brooks is my first lab buddy, who also joined in the same semester, and advised by Marc. I do not remember how many hours we worked together on projects, how many times we pulled an all-nighter for a programming assignment. But I do really enjoy those moments, when both of us were innocent and only kept exploring without worry. Other graduate students, faculty and staff members at UIUC that I could not name all, they played a very important part of my life, and time to time, were the source of inspirations for me to keep moving on in my research. Thank you all!

Not all graduate students were like myself. I managed to get married and had a child at the time of my graduation. Some said it was challenging, some said it gave much motivation, both are true. My dear wife came to me at the time I needed most. She helped get my personal life organized and even gave me a gift, my dear son. My parents and my sister are always understanding and supportive to me regardless of any life decisions I made. I cannot thank them more. Family is not an important thing, it is everything to me.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Communication is a fundamental property of parallel computing. Communication can happen at every layer: from the memory to caches or registers, across the network to a remote memory location. Among these types, communications that are not part of a sequential program such as across the network are considered parallel overheads. Thus, researchers in parallel computing have devoted significantly on minimizing these overheads through communication-optimal algorithms [1, 2, 3, 4], as well as trading it with redundant computations using communication-avoiding techniques [5, 6, 7],

Once an algorithm is selected, programmers write program to map it to the machine model, such that its execution is correct and the performance behavior is as predicted by the model. This is however, often challenging due to the large scale of the system and the complexity in their hardware architecture. Hence, research in programming model and its efficient implementations are also important to ensure this transition is effective and error-free.

This dissertation focuses on these layers, in particular, the communication system, its programming model and implementations. It begins by analyzing limitations of message-passing interface (MPI) [8], the current state-of-the-art system for programming parallel applications, with respect to both implementations and semantics, and provides with solutions.

The dissertation also introduces the concept of *efficient concurrent message-passing* as a method for accelerating the communications further beyond MPI. The Light Communication Interface (LCI), an implementation of such concept enables Graph frameworks to achieve high performance on distributed-memory and heterogeneous architectures. It opens up opportunities for further improvement on emerging applications and frameworks.

1.1 EXPRESSING COMMUNICATION

An important aspect of the programming model for parallel computers is how to express communications in a parallel program. One way is to use the notion of producer-consumer relationship: a producer is one who owns or produces the data via prior computation, likewise, the consumer is one who would like to make use of the data for its own computation. How the data is moved from producers to consumers can be described either implicitly or explicitly by a programmer and by the programming system. The communication is explicit when the programmer must write their intention for communication in the program code, like an instruction or a function call. For example, in the message-passing model, the producer must specify a SEND, and a consumer must specify a RECEIVE, both with a buffer as one of the arguments so that the lower layer knows that the data should be moved from one to the other and executes that. Otherwise, the communication is considered implicit.

An example of implicit communication is in parallel algorithms written in shared-memory machine. Since data are shared between computation units, any instruction executed in any unit can read or write directly to a shared memory location. The order of read and write is dictated by memory consistency model of the machine [9], memory cache and cache coherence system maintains both performance and correctness [10]. To express parallelism, the notion of threads is introduced, which informally is a path of the program that can be executed by a different computation unit of the machine. Communications happen when threads share variables i.e. one thread writes data that later be read and use by another thread. To ensure the correctness in data sharing including preventing data races, synchronization primitives are developed (such as mutex, semaphore, monitor, condition variables [11, 12, 13]).

Implicit communication model does not scale well. Aside the complexity of concurrent data-structure and synchronization primitive, the implicit communication between hardware units places a burden in the hardware design, due to the needs for maintaining an efficient cache coherency and memory consistency model. For these reasons, modern large-scale parallel machine are built with distributed-memory, and programmed using explicit communication models [14]. Message-Passing Interface (MPI) is a well-known example, in which the communication is expressed directly by an application.

1.2 MESSAGE-PASSING INTERFACE

Since coming to existence, MPI has become the de-facto programming interface for high-performance applications running on supercomputers. MPI is theoretically rooted in the Communication Sequential Process (CSP) [15], a mathematical model for concurrency. An important aspect of the model is the notion of share-nothing i.e. the model consists of n concurrent processes having no shared variables or common memory addresses. As a consequent, all interactions between processes is done via message-passing.

CSP is however only an abstract model i.e. It does not provide any insight on how it can be implemented. Whereas, MPI focuses on high-performance implementation. MPI extended CSP with many other modern parallel programming features including collective communications, or data-type manipulations. This is one of the key for its success. MPI provides a rich set of application programming interface (APIs) that matches many types of producer-consumer problems. MPI implementation is generally the most portable and well performing for many types of parallel architectures. Any amendment to the MPI standard is continuously reviewed and evolved by many experts in the fields.

MPI has stayed as the dominant programming model of supercomputers for roughly 10 years, until It faces new challenges of the Post Moore's Law. New CPU architecture comes with many processor cores, for both low and high-end users. Parallel computers although stay as distributed-memory architecture, start to be built with more cores per compute node. In addition, the emergence of accelerators and special-purpose hardware, adds the need for communication between these units within a machine and remote machine.

How do a programmer deal with such a hybrid machine? MPI researchers come up with two answers: one is called the *MPI-everywhere*, and one is called *MPI+X* model. Other researchers prefer application frameworks.

1.3 MPI AND APPLICATION FRAMEWORK

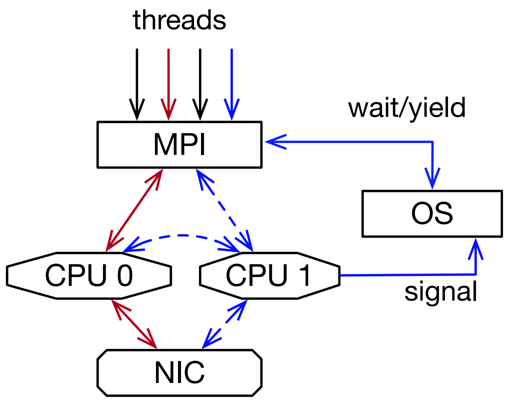


Figure 1.1: When a single thread (red) accesses MPI, it can go directly to the network; if more threads try to access (blue), one may be descheduled back to the OS scheduler; the execution also loses locality by moving to another processor core.

The "MPI-everywhere" says that one could just use the same MPI program, assigning one MPI process per processor core. The same program written in MPI can then work out of the box. Thereby, removing the burden of a programmer from having to update his code. The burden is however, moved to the MPI implementations, such that it has to handle much more number of processes (can be easily a billion in unit) and many of which can be different in nature. For example, a process can be local in the same compute node, or remote across the network. MPI implementation has made fast path for such cases [16, 17]. MPI implementations has also tried to map MPI rank to OS threads [18, 19, 20] to reduces memory consumption. However, a process can also be associated with an accelerator or special purpose hardware. It is an open question how this can be implemented efficiently in a MPI implementation.

The ”MPI+X” model takes a different path, which places harder burdens on both application programmers and MPI implementations. It however reduces pressure on scalability and portability. In this model, MPI is only used for communications across the network. Within a compute node, the native programming for the particular hardware is used (e.g. multi-threading for multi-core CPU). The obvious burden of the programmer is that they need to partition the problem in various ways to fit each type of hardware and in many cases, must rewrite the original MPI program. Figure 1.1 depicts a situation when multiple threads trying to access MPI.

There is in fact, no successful applications prior to the study of this dissertation when MPI is used in combination with shared-memory multi-threaded parallelism (MPI+Thread) model. There are efforts in the implementation such as Pioman [21], or MPI optimizations [22], but the usage is limited and the performance is still not matching single-threaded communication style.

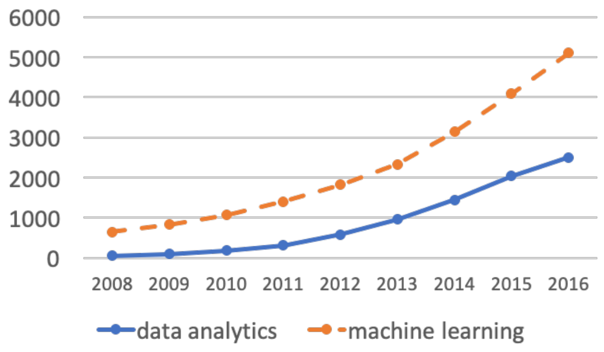


Figure 1.2: The number of scientific papers having HPC and ”data analytics” or ”machine learning” keywords

While there is no clear winner between the two aforementioned models, new set of applications are being developed as the entirely new users of supercomputers: graph, data analytics, machine learning. One evidence is demonstrated in Figure 1.2 where usage of HPC increases every year in these domains. The reason is the availability of a large amount of data and the needs for computations to process them in a short amount of time. Applications of these domains are written in their existing frameworks; and the frameworks extend its capability to handle distributed-memory and heterogeneous environment. A noteworthy feature during the transition is the adaptation of MPI for transferring data across network [23]. MPI becomes a portable communication middleware, which is however not an ideal option.

The first goal of MPI was designed for written directly by the application, secondary is library or frameworks [24]. Layers of software abstractions, each has its own semantics and not necessarily fit well together, can hurt performance. MPI is generic, but the generalization comes with extra costs. It becomes clear towards the end of the dissertation that a specially-designed communication interface is the answer for such development.

1.4 OTHER APPROACHES

One alternative for MPI in modern super-computing domain is the Active Messages (AM) model [25], with the GASNET implementation [26]. GASNET focuses on optimizing the ability for doing Remote Direct Memory Access (RDMA) in the modern network interface. GASNET targets the Partitioned Global Address Space (PGAS) programming languages like Chapel [27]. Most of the issues that GASNET brings up due to insufficient RDMA support in MPI is however resolved with MPI 3.0 using MPI one-sided communication. GASNET remains a good library for Active Messages model, despite there is no modern hardware that can support Active Messages efficiently. In addition, same as MPI, GASNET does not support good interaction between threads and communication runtime.

GASPI [28] is a communication library that provides a flexible interface focusing on efficient support for one-sided remote notification. GASPI argues that such primitive is sufficient for efficiently support both of both existing MPI and PGAS applications by retargeting them towards asynchronous dataflow model. Our approach takes advantage of this design and arguments, while realizing the need for extending the capability to other generic communication patterns. In particular, efficient notification should be supported for efficient producer-consumer synchronizations, but can be extended beyond one-sided communication with minimal overhead.

OpenFabrics libfabric (OFI) [29] and Unified Communication X (UCX) [30] are two recent industry-driven effort for unifying HPC networking middle-where. These libraries provide one-sided communication that allows implementing PGAS-style RMA, in addition to tag matching and messaging queues suitable for implementing message-passing APIs. Users of OFI and UCX are existing communication model such as GASNET, OpenSHMEM, MPI. This provides more portability support, but the performance optimization for applications are still limiting by layers of abstractions provided by these models. It is not clear OFI and UCX plans to support for other frameworks, and initial experiments show they do not support multi-threading well.

1.5 LIGHT COMMUNICATION INTERFACE

The Light Communication Interface (LCI) is a highly customizable communication interface that targets efficient concurrent message-passing model. Its goal is to allow an efficient interaction between the application layer or the framework layer and the communication layer. It considers in its design the entire stacks of communication software: from when the producer executing a communication call, to when the data is available and the consumer is notified. The goal of LCI design is such a path should have a constant additional overhead even with extreme cases when millions of producers and consumers are communicating concurrently.

The ideas of LCI come from a thorough study of MPI implementations and semantics with respect to emerging multi-core architecture and programming model including hybrid programming and application framework. LCI is designed from the ground up with all the lessons learn from MPI implementations. Hence, LCI can keep some goodness of MPI while ignoring features that cannot perform well. Since LCI feature sets are smaller and carefully chosen to match current hardware and selected emerging frameworks, LCI performance also matches the native performance of the hardware much better while providing a flexible programming environment.

LCI's approach enables a new paradigm: threads within a compute node can communicate concurrently to a remote entity without losing any performance. In some cases, such style of communication can even accelerate the overall throughput. This paradigm is called efficient concurrent message-passing.

When communication becomes the major overhead, accelerating its performance by exposing and allowing independent communication is desirable. Especially, when current hardware can support multiple independent communication channels. To our best knowledge, LCI is the first open-source communication library that can expose this type of parallelism in communication, in a way that is efficient and useful.

1.6 DISSERTATION OVERVIEW

This dissertation begins in Chapter 2, with a case study of a MPI implementation – the MPICH library. It exposes several problems limiting the performance of MPI for concurrent accesses which prevents MPI to be used efficiently with shared-memory programming via multi-threading. The solutions show that for MPI to be inter-operate with other programming model, or in other words, for MPI + X model to work, a significant amount of engineering is required to refactor the existing MPI implementation.

Chapter 3 pays attention to the low-level communication in distributed-memory graph analytics framework, one of the emerging new application for high-performance computing. It gives an insight that programming the communication using MPI directly is not the ideal choice; several mismatches are found, which can be resolved by a specially-targeted communication library that matches better to both hardware and applications.

Chapter 4 discusses inter-operations between communication library and multi-threading library. It shows that this is a special type of producer-consumer synchronization that can be accelerated by a novel thread scheduler, targeting this pattern. The overall results show that, the synchronization between the communication layer and the thread layer can be as simple as a bit-flip, which can be implemented directly in hardware. This result advocates a tighter integration between these layers for a better performance.

Chapter 5 devotes to the core definitions and implementations of the Light Communication Interface (LCI), a communication library that targets efficient concurrent communication message-passing and application-specific communication runtime. It argues the semantics of MPI message-matching should be relaxed; which allows much higher concurrency to the communication runtime and reduces the performance significantly.

Chapter 6 concludes the dissertation with a summary of our works and a discussion on how LCI and concurrent communication can be used to further accelerate the communication of applications and frameworks.

CHAPTER 2: MULTI-THREADED COMMUNICATION IN MPI

Message Passing Interface (MPI) standard defines four threading modes with increasing degree of interaction between application threads and MPI i.e. `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. For instance, if the application never uses thread, `MPI_THREAD_SINGLE` is used, and MPI implementation can take a faster path without the need of thread-safety assurance. Most of the application nowadays, makes use of either `MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED` where multiple threads are used, but only one thread may access the MPI calls at a time.

Our focus is on the `MPI_THREAD_MULTIPLE`, which allows flexible and concurrent multi-threaded MPI calls. This mode has been gaining more attraction in the application space. The primary driving factors are ease of programmability for emerging fine-grained threading models and the desire to efficiently utilize modern network fabrics that requires multiple communicating cores to fully exploit their capabilities. In order to meet this expectation, thread-safety is a prerequisite, and its corresponding overheads should be minimal.

Unfortunately, the current state-of-the-art thread-safety support in MPI implementation is far from ideal. Most production implementations satisfy the core of the thread compliance through locks, since using exclusively lock-free or wait-free objects is difficult to implement correctly and to maintain. MPI implementations were shown to suffer significantly from scalability issues due to lock contention. Different aspects of locking in MPI implementations have been studied in the literature. Lock granularity were explored in [31, 32], reducing lock ownership passing latency has been extensively studied [33, 34, 35], locking arbitration is studied in [36]. All previous works however use simple heuristic to decide which thread is allows to acquire the lock, thus suffer from high contention and cache trashing.

This chapter studies the current affair of threaded communication using MPICH [37], a production quality MPI that has several derivation (such as IntelMPI [38], MVAPICH2 [39]). Two advanced synchronization techniques for improving the implementation will be introduced and evaluated through several benchmarks and production kernels.

2.1 ANALYSIS OF THREAD-SAFETY ISSUES IN MPICH

One reason for MPICH success is the multi-layered design, which allows it to be both efficient and portable with respect to many low-level communication drivers. At the time of study, MPICH is the only MPI production implementation that fully provides stable support for all threading mode of the MPI standard. However, several of previous works have raised the issue of thread-safety in MPICH, mainly its performance with `MPI_THREAD_MULTIPLE`.

MPICH is a lock-based MPI implementation where function calls are protected through the use of locks. MPICH has a single critical section (CS) (except for MPICH/BGQ where a few finer-grained locks are used), and maintained a coarse-grained lock (*global lock*) to protect it. In this design, *arbitration* of the concurrent accesses to the CS is an important factor.

Suppose that one thread is performing a blocking operation (e.g., `MPI_Recv` or `MPI_Wait`) and successfully acquires the global lock of the MPI library. If the operation cannot be satisfied immediately, the thread **must** release the lock. Failure to do so prevents other threads from entering the CS, violates MPI progress requirements, and may lead to a deadlock. Therefore, arbitrating lock ownership has a correctness implication.

In the remainder of the chapter, all presenting thread-safe methods guarantee an arbitration correctness, and the focus is on the performance implication.

We illustrate the relationship between thread safety and MPI communication in Figure 2.1. It describes a simplified implementation of `MPI_Isend` and `MPI_Wait`. These routines are examples of a nonblocking MPI call, to send a message, and a blocking MPI call, to wait for its completion.

In the example, we distinguish two major code paths with distinct progress properties. The first, which we refer to as the *main path*, is taken by both routines between the first lock acquisition (lines 18 and 40) and the last lock release operation (lines 22 and 44). This code path is similar to most MPI routines and often advances the system.

The other path, which we refer to as the *progress loop*, concerns only blocking calls. It is characterized by a tight loop (line 28) waiting for the completion of a target operation. This path does not guarantee progress and is characterized by high-frequency lock acquire/release operations. Thus, by our prior definition, threads executing the progress loop are waiting threads and the others are active threads. *We consider lock acquisitions by waiters as wasted (or simply as causing waste) when they yield no progress while active threads are waiting for the lock.*

Regardless of the state in which a thread is, the arbitration of the concurrent accesses is dictated solely by the lock arbitration. That is, lock ownership passing defines the order in which threads execute the CS.

In order to promote progress of the system, previous works exploited this distinction between threads in the locking implementation [36, 40]. Considering active threads as having a higher priority; the traditional lock acquisition can be extended with a low or high-priority interface. The goal was to have waiting threads acquire the lock with lower priority than that of active threads (line 33), to inject more useful works into the system.

```c
typedef struct request {
  REQUEST_BODY;
  bool complete;
  void *(*completion_cb)(*);
} request_t;

// Global lock
lock_t g_lock;

// Callback to complete a request
void complete_request(request_t *req) {
  req->complete = true;
}

// Issuing a nonblocking send operation
void MPI_Isend(..., MPI_Request *handle) {
  request_t *req;
  acquire(g_lock);
  req = create_request();
  req->completion_cb = complete_request;
  Isend_body(req);
  release(g_lock);
  set_request(handle, req);
}

// Internal progress routine
void progress_wait(request_t *req) {
  while (!req->complete) {
    bool made_progress = poll_network();
    if (!made_progress) {
      release(g_lock);
      yield();
      acquire_low(g_lock);
    }
  }
}

// Waiting for request completion
void MPI_Wait(..., MPI_Request *handle) {
  acquire(g_lock);
  request_t *req = get_request(handle);
  progress_wait(req);
  *handle = MPI_REQUEST_NULL;
  release(g_lock);
}
```

Figure 2.1: Simplified thread-safe implementation of `MPI_Isend` and `MPI_Wait`. It assumes a global CS (protected by `g_lock`) and a callback-based request completion. `poll_network` is a hardware network call that progresses all outstanding operations. In particular, during this call, user-provided completion callback functions are executed when the corresponding operations have completed. `complete_request` simply marks the request as complete. Lock ownership passing is ensured through the `acquire`, `acquire_low` (for low priority), and `release` calls.

By combining the priority method with FIFO-ordered locks (ticket [41] and CLH [42]), better communication progress can be achieved while maintaining correctness. This solution, however, relies on a $\mathcal{O}(N)$ FIFO-ordered lock-passing operations to find an active thread.

2.1.1 Shortcomings of Existing Methods

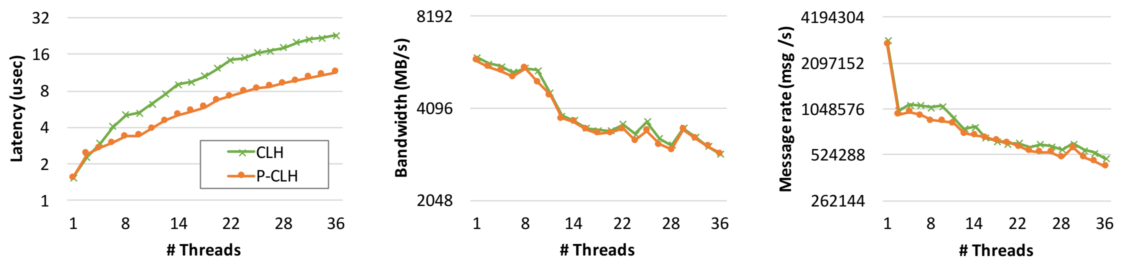
This section re-evaluates the issues in the multithreaded communication and describes the motivation for our methods. For clarity, we define the following terminology:

- MTX: MPICH implementation using a Pthreads mutex as a lock (this is the same as the MPICH 3.2)
- CLH: MPICH implementation using the CLH lock [40]
- P-CLH: MPICH implementation using the CLH lock with prioritization of the main path [40]

We use latency, bandwidth, and message rate as metrics to evaluate the performance of the baselines CLH and P-CLH. We note that the performance of these locks are superior to that of MTX for `MPI_THREAD_MULTIPLE` when there is no oversubscription [40].

The latency benchmark is similar to the multithreaded OSU latency benchmark (`osu_latency_mt` [43]) except that we used OpenMP instead of Pthreads to take advantage of the OpenMP thread-binding capability. This benchmark uses two MPI ranks, one per compute node. One MPI rank (*sender*) performs a number of pairs (10,000 for messages up to size $\leq$ 8 KB, 1,000 for larger messages) of `MPI_Send` and `MPI_Recv` to the other MPI rank (*receiver*). The receiver creates a fixed number of threads to perform a corresponding `MPI_Recv` and `MPI_Send` such that the total number of requests matches the sender's. The latency is computed at the sender by taking the average time used per message.

The bandwidth and message rate benchmarks are the same as those studied in [36]. These benchmarks also create two MPI ranks in two different nodes (sender and receiver); each creates the same number of OpenMP threads. The sender performs a number of `MPI_Isends` in each thread (the default 64 messages is used in our experiments) before waiting for all of them with `MPI_Waitall`; It then performs `MPI_Recv`. The receiver side similarly performs `MPI_Irecv`, `MPI_Waitall`, and `MPI_Send` in each thread to match the sender. This process is repeated so that the number of messages is the same as that in the latency benchmark. The message rate is computed at the sender as the total number of messages over the overall execution time. The bandwidth is computed similarly but using the total size of transferred data instead of the number of messages.



(a) Latency (the lower, the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

Figure 2.2: Initial performance results of CLH and P-CLH. The message size is 64 bytes for the latency and message rate benchmarks and is 1 MB for the bandwidth benchmark.

Our platform for these experiments is a cluster of Intel Haswell-EP machines. Each node consists of two Intel Xeon E5-2699 v3 CPUs (36 cores in total) whose cores are arranged into four NUMA domains. The nodes are interconnected by using Mellanox FDR InfiniBand. We compiled our benchmarks and MPICH (for both CLH and P-CLH) using the Intel compiler 16.0.3 with the MXM low-level communication runtime included in HPC-X 1.6.392.

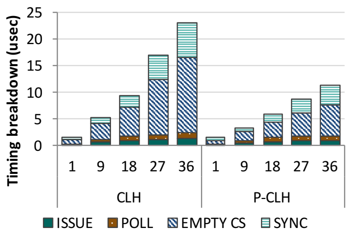
We used the `MPI_T` instrumentation available in MPICH for profiling and instrumentation, in order to obtain breakdowns in timing and internal counters such as the number of network polls, and HPCToolkit [44] for measuring the number of cache misses. All our tests were done with each thread bound to a CPU core using two OpenMP environment variables: `OMP_PROC_BIND=close` and `OMP_PLACES=cores`.

MPICH uses a single communication context per MPI process; thus, we expect high degrees of contention for communication-intensive codes. Furthermore, message rates are bound by single-threaded performance because of the limited concurrency of a globally locked single-communication context implementation.

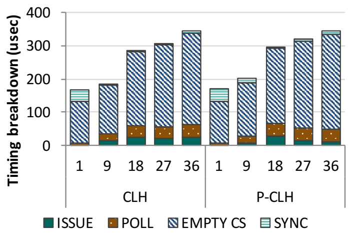
Figure 2.2 illustrates the results of each benchmark. The results show that as the number of threads increases from 1 to 36, the communication performance degrades significantly. For instance, while the latency measured with one thread is about 1.5 μs, it increases to 23 μs for CLH and 11.5 μs for P-CLH with 36 threads. In the following subsections, we analyze the performance of each benchmark in detail.

2.1.2 Analysis of Latency and Bandwidth Results

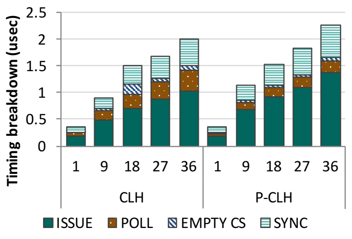
To reveal where most of the execution time in Figure 2.2 is spent, we divided the execution time into four parts as shown in Figure 2.3. The breakdown timing is measured and reported



(a) Latency (the lower, the better)



(b) Bandwidth (the higher, the better)



(c) Message Rate (the higher, the better)

Figure 2.3: Execution time breakdown of the performance results in Figure 2.2. ISSUE, POLL, EMPTY CS, and SYNC represent time spent in issuing operations, time spent in making progress, time wasted in the CS without doing useful work, and time spent in synchronizations, respectively. The timing overhead is about 8%, 1% and 6% on average (harmonic mean) of the total execution time for the latency, bandwidth and message rate benchmark respectively.

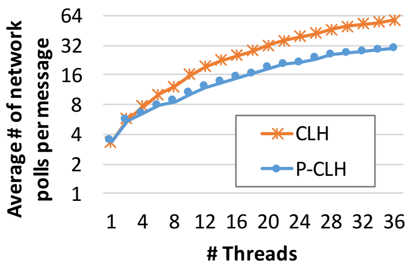
at the sender side (similar results can be obtained at the receiver side).

In the figure, as the number of threads increases, the execution time of each portion also increases. Most notably, in the latency and bandwidth cases, we see a large increase in EMPTY CS, that is, wasted time in the CS without doing useful work.

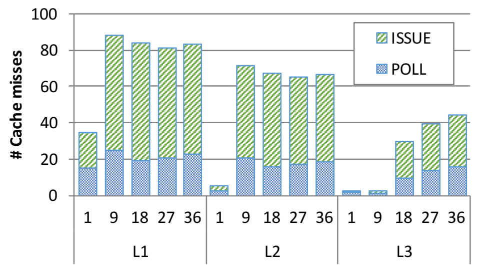
The waste in the latency and bandwidth results is caused primarily by the increased number of network polls when more threads are involved in the communication.

Figure 2.4(a) shows the average number of network polls performed per message in the latency benchmark (we omit the result of the bandwidth benchmark because it showed a similar pattern). Although the number of messages is constant across all experiments, the number of network polls per message increases as the number of threads increases. Recall that any waiting threads executing a blocking operation can enter the CS through the progress loop. If more waiting threads enter the CS and perform unnecessary work (e.g., network polling), it can delay active threads from proceeding and consequently slow the overall performance.

By prioritizing the main path, P-CLH enables more work to be injected into the runtime compared with CLH; hence it improves the performance because of the presence of more active threads. However, P-CLH suffers from $\mathcal{O}(N)$ FIFO ownership passing between N waiting threads, which reduces its performance with increased numbers of threads.



(a) Average number of network polls performed per message in the latency benchmark



(b) Average number of L1, L2, and L3 cache misses per message in `MPI_Irecv/Isend` (ISSUE) and in `MPI_Wait` (POLL).

Figure 2.4: Analysis of different components in latency, bandwidth and message rate with MPI using CLH and P-CLH locks

.

2.1.3 Analysis of Message Rate Results

The execution time breakdown of the message rate results in Figure 2.3(c) shows that the majority of its time is spent in issuing messages (i.e., ISSUE). The overall message rate is again confirmed to be bounded by the issuing rate, as also reported in [40].

This phenomenon can be understood better by looking at the number of cache misses in Figure 2.4(b). As we increase the number of threads in the experiment, the number of cache misses increases substantially—first in the L1 and L2 caches within the same NUMA node (≤ 9 threads) and then in the L3 cache when going out of the NUMA node (> 9 threads). This increasing rate reflects the trend of our message rate benchmark results in Figure 2.2(c).

The increase in cache misses happens mainly in the issuing part of the benchmark because of the FIFO ownership passing exercised in the CLH lock used in CLH and P-CLH. Even though issuers initiate MPI operations and make progress, they take turns in entering the CS, thus causing the data movement and resulting in the increase in cache misses. The decreasing message rate, with a larger number of threads, signifies the problem of thread synchronization, especially lock ownership passing, between active threads.

From the analysis of these benchmarks, we have identified the major sources of the performance loss:

- The spurious wake-ups of threads such as in the latency benchmark. We solve this using an algorithm for waking up threads based on the status of the communication.
- Cache misses and lock conflicts. We solve this using a better lock implementation featuring both priority and data locality.

2.2 MESSAGE-DRIVEN COOPERATIVE THREAD SCHEDULING

Our idea for solving the latency and bandwidth problems described in Section 2.1.2 is to allow only one waiting thread to drive a communication context while making all other waiting threads wait outside the CS until their request is completed.

At most one thread, called the *server*, is elected among all waiting threads, and only the server is allowed to enter the CS and poll the network for communication progression. In order to support the model of progress in MPICH, and easily integrated in existing MPI implementations, we do not assume a centralized entity for making progress or thread arbitration, but design a decentralized strategy. Restricting access to a single waiter also increases the likelihood of residency in cache of the communication context data structures and avoids contention for the CS.

Remaining or new waiting threads become *waiter*s and wait until the server completes their request. A waiter indicates its waiting intent through a *synchronization counter* whose initial value equals the number of pending operations that it is waiting for. When the server completes a request, it enables the waiter associated with the request to continue the execution with a signal (i.e., work-driven, selective reactivation).

To avoid starvation, when the server finishes its own request, it elects another waiter, if there is one, to hand over its server role; otherwise, the waiters will be waiting forever, since none is making progress. Note that the number of servers depends on the network hardware and communication volume. For our purpose, a single server is sufficient; but the selective reactivation method can easily be extended for multiple servers. Figure 2.5 depicts our selective reactivation method with a single server.

Implementing the selective reactivation requires two important changes to the runtime: (1) defining a synchronization counter implementation and (2) associating a counter per blocking operation and storing its reference in the corresponding request objects. In Figure 2.5, we assume a generic synchronization counter object referred to by the abstract type `scount_t`.

This object supports two types of operation: a signal operation is translated to `scount_-signal()`, and a wait operation is translated to `scount_wait()`. Each request contains a `scount_t` object (line 5) representing our synchronization structure. For an `MPI_Waitall` operation, a reference to the counter object will be initialized to the number of pending operations and stored in all of them to consume signal events.

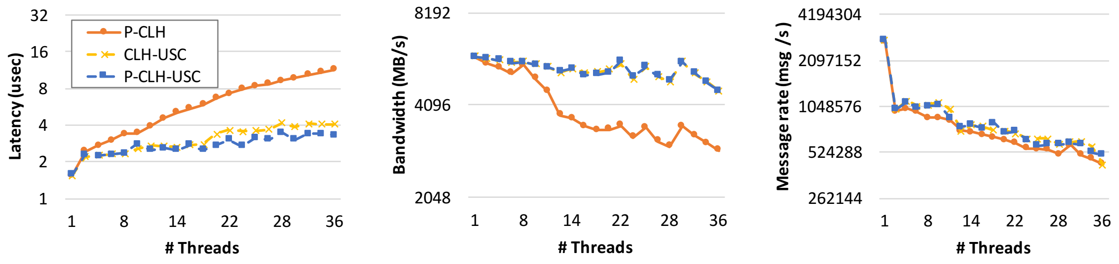
Since a request object is associated with a `scount_t` object, we can tie the waiting thread to the request by making the waiting thread wait on the associated `scount_t` object. In addition, because the one-to-one mapping between a thread and a `scount_t` object is maintained, one `scount_t` object can be regarded as representing one waiting thread. If the

```
typedef struct request {
  REQUEST_BODY;
  bool complete;
  void *(*completion_cb)(*);
  scount_t scounter;
} request_t;

bool server = false; // is there a server?
list<scount_t *> waiters; // list of waiters

// Callback when a request is finished
void complete_request(request_t *req) {
  req->complete = true;
  list_remove(waiters, &req->scounter);
  scount_signal(&req->scounter, false);
}

void progress_wait(request_t *req) {
  bool elected = false; // am I the server?
  while (!req->complete) {
    bool made_progress = poll_network();
    if (!made_progress) {
      if (!server) {    // no active server
        elected = true; // I am elected
        server = true;  // as a server
      }
      if (elected) { // I am a server
        release(g_lock);
        yield();
        acquire_low(g_lock);
      } else {       // I am a waiter
        list_append(waiters, &req->scounter);
        scount_wait(&req->scounter,g_lock,1);
      }
    }
  }
  if (elected)      // I am no longer
    server = false; // the server
  // Wake up a potential server
  if (!server && !list_empty(waiters))
    scount_signal(list_pop(waiters), true);
}
```

Figure 2.5: Modifications to Figure 2.1 for selective reactivation.

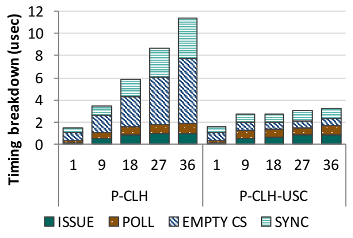
```
typedef struct scount {
  int     cur_count;
  cond_t cvar;
} scount_t;

/* Wait for N events. This routine assumes
   the lock L associated to the condition
   variable C->cvar is held. */
void scount_wait(scount_t *C, lock_t L,
                 int N) {
  C->cur_count = N;
  /* cond_wait releases L to wait for signal,
     then reacquires it at returns. */
  cond_wait(&C->cvar, L);
}

/* Signal one event or force-wakeup on
   condition variable C->cvar. */
void scount_signal(scount_t *C, bool force) {
  if (force) cond_signal(&C->cvar);
  else {
    C->cur_count--;
    if (!C->cur_count) cond_signal(&C->cvar);
  }
}
```

Figure 2.6: Example implementation of a *synchronization counter*. It assumes the existence of generic condition variable (`cvar`) and lock (`L`) implementations. The routine `scount_wait` allows waiting for `N` events and `scount_signal` decrements the number of pending events (`cur_count`) for `C` and wakes up the corresponding thread when there are no more pending events. We also allow forced wakeup with the Boolean `force`.

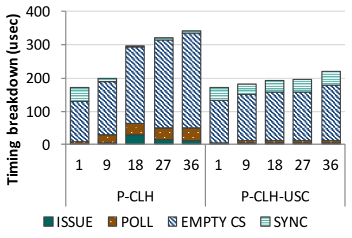


(a) Latency (the lower. the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

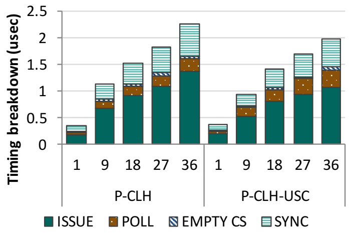
Figure 2.7: Performance results of CLH and P-CLH with selective reactivation. The message size is 64 bytes for the latency and message rate benchmarks and is 1 MB for the bandwidth benchmark. CLH-USC and P-CLH-USC represent the results using selective reactivation with a user-level synchronization counter implementation along with CLH and P-CLH, respectively.



(a) Latency (the lower, the better)

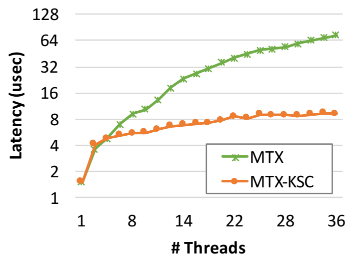


(b) Bandwidth (the higher, the better)

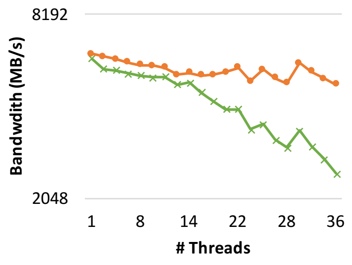


(c) Message rate (the higher, the better)

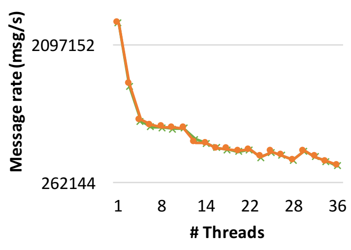
Figure 2.8: Execution time breakdown of the P-CLH and P-CLH-USC results in Figure 2.7. We omit the breakdown of the CLH-USC results because it shows a pattern similar to that of the P-CLH-USC results. ISSUE, POLL, EMPTY CS, and SYNC represent time spent in issuing operations, time spent in making progress, time wasted in CS without doing useful work, and time spent in synchronizations, respectively. The timing overhead is about 8%, 1%, and 7% on average (harmonic mean) of the total execution time for the latency, bandwidth, and message rate benchmark respectively.



(a) Latency (the lower, the better)



(b) Bandwidth (the higher, the better)



(c) Message Rate (the higher, the better)

Figure 2.9: Performance results of the vanilla MPICH-3.2 (MTX) and our modification using selective reactivation with the Pthreads mutex and kernel-level synchronization counter (MTX-KSC). The message size is 64 bytes for the latency and message rate and is 1 MB for the bandwidth benchmark.

thread cannot complete the request and is required to wait, it will not enter the CS using the *progress loop* path but instead will be blocked inside the synchronization object if a server already exists, as shown at line 33.

When the request is finished, the signal operation is performed by the server at the callback function (line 15). The `scount_t` objects are linked in a globally shared doubly linked list (line 9), so that the server can find a target waiter when needed. With the doubly linked list, we can efficiently remove a waiter from the list when it is signaled, that is, when the request is finished by the server.

An example implementation of the synchronization counter is illustrated in Figure 2.6. It assumes generic lock and condition variable implementations underneath. A `scount_wait` operation publishes the intent to wait for `N` events. `scount_signal` events decrement the counter and wake up the thread when reaching zero (i.e., all events are satisfied). The `force` argument forces a thread wakeup regardless of the counter value. This is essential for electing the next server when the current one is leaving the runtime.

In the case of waiting for the completion of multiple requests, more than one request object may share one synchronization counter object. The server issues signals whenever it completes one of the requests, and a thread wakeup is triggered when the counter has reached 0 (see line 15 in Figure 2.5 and the routine `scount_signal` in Figure 2.6). This strategy prevents the waiter thread from being woken up multiple times, many of those just to figure out that not all its requests are finished, and thus adding meaningless synchronization overhead.

To support other blocking operations without requests such as `MPI_Win_flush`, we can tie the waiting thread to associated pending objects, for example, `MPI_Window`, in a similar way. We leave for future work the task of extending our selective reactivation method to support these kinds of operations.

We implemented the selective reactivation method in the baseline CLH and P-CLH. Figures 2.7 and 2.8 illustrate the performance results of our implementations with the microbenchmarks and the execution time breakdowns, respectively. The results show that compared with the baseline, our selective reactivation method is effective in lowering the communication latency in multithreaded communication cases (Figure 2.7(a)) and in improving the bandwidth significantly with a larger number of threads (Figure 2.7(b)). Specifically, the latency and bandwidth are improved by 3 times and 5 times with 36 threads, respectively. These improvements come mainly from the reduction of the time spent in EMPTY CS, as shown in Figures 2.8(a) and 2.8(b).

The selective reactivation method can also be applied to MTX by using a kernel-level synchronization counter, which can be implemented with a Pthreads condition variable and a counter. However, since the kernel-level synchronization counter has higher latency due to its kernel-specific implementation, it should be used only when oversubscription is required. The performance improvement when applying the selective reactivation technique to MTX is shown in Figure 2.9.

The selective reactivation (denoted as MTX-KSC) performs significantly better than the baseline in the latency and bandwidth benchmarks. However, the absolute latency of MTX-KSC is still far worse than that of CLH and P-CLH using selective reactivation (P-CLH-USC and P-CLH-USC in Figure 2.7(a)) because of the higher latency of both the kernel-level lock and synchronization counter implementations.

2.3 MUTUAL EXCLUSION AND UNBOUNDED-BIASED LOCK

In the preceding subsection, we showed that our selective reactivation technique can eliminate wasted executions and ensure that any thread entering the CS has a high chance of performing useful work. However, managing the thread scheduling alone does not improve the performance of the message rate benchmark (Figure 2.8(c)). The reason is that message rates were bound by the injection rate of nonblocking calls that suffers from intranode data movement, as was analyzed in Section 2.1.3. This section proposes a new locking strategy that is designed to reduce cache misses.

First, we introduce a *locality-preserving locking with unbounded bias* method based on CLH, which we refer to as CLHub. The lock is designed to provide the necessary property for our purpose — it is biased toward the high-priority thread that most recently released the lock.

The key idea is to combine a simple compare-and-swap spin-lock lock (i.e., a biased lock) to exploit the lock monopolization with two FIFO locks to handle high and low priorities and providing the fairness property when needed (e.g., reducing thread contention on the lock by queuing threads in the lock structure).

In Figure 2.10, the data structure for the lock includes the following fields (lines 1–6): `bias` – a spin lock that has a biased behavior; `fifoH` and `fifoL` – CLH locks to block threads in the high-priority and low-priority paths, respectively; and `filter` – a flag to switch between two priorities.

This lock allows only high-priority threads (i.e., active threads in our case, which call `acquire()`) to utilize the monopolization, while low-priority threads (i.e., waiting threads) rely on the FIFO property of the lock without the monopolization through `acquire_low()`.

Since active threads always advance the system, in addition to raising their priority over waiting threads, we synchronize their concurrent accesses with a locality-preserving high-

```
typedef struct clhub {
  spin_lock_t bias; // biased lock
  clh_t fifoH; // FIFO lock for high priority
  clh_t fifoL; // FIFO lock for low priority
  int filter;  // to switch between different priority paths
} *clhub_t;

void acquire(clhub_t l) {
  if (try_acquire(l->bias) == fail) {
    acquire(l->fifoH);
    l->filter = 1;
    acquire(l->bias);
    l->filter = 0;
    release(l->fifoH);
  }
}

void acquire_low(clhub_t l) {
  acquire(l->fifoL);
  while (l->filter == 1) {
  }; // busy wait.
  acquire(l->bias);
  release(l->fifoL);
}

void release(clhub_t l) {
  release(l->bias);
}
```

Figure 2.10: Pseudocode of the locality-preserving lock with unbounded bias, which uses a combination of a spin lock and two CLH locks. For simplicity, we use the same function names (`acquire` and `release`) for different lock types, but please consider them as overloaded functions that can handle proper lock types.

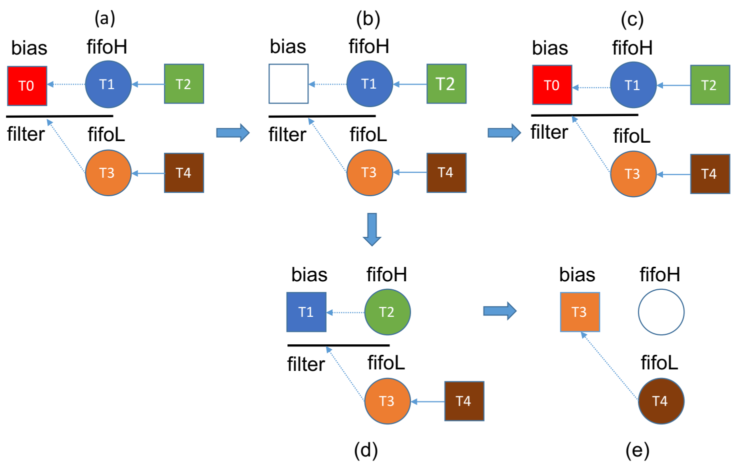


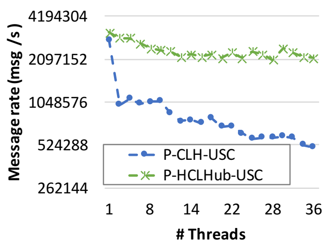
Figure 2.11: Illustration of five threads (T0,..,T4) using the locality-preserving lock: (a) T0 performs `acquire()` and succeeds, taking `bias`. Right after T0, T1 and T2 perform `acquire()` while T3 and T4 perform `acquire_low()`. As a result, T1, T2, T3, and T4 spin at `bias` (i.e., T1 becomes *candidate*), `fifoH`, `filter`, and `fifoL`, respectively. (b) T0 releases `bias`. (c) If T0 calls `acquire()` immediately again, it has a higher chance of taking `bias` than does T1 because of locality. (d) If T0 finishes and moves on, T1 can succeed, acquiring `bias`. (e) Only when there is no high-priority thread is `filter` released and low-priority threads can acquire `bias`.

throughput lock. The locality preservation is achieved through a competitive ownership passing, which results in core-level unbounded lock monopolization. The monopolization achieves locality preservation but does not cause starvation for waiting threads since active threads are guaranteed to complete their operations in a bounded number of steps.

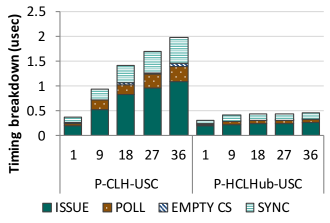
Figure 2.11 illustrates an example usage of our locality-preserving lock of Figure 2.10. In the figure, threads T0, T1, and T2 perform `acquire()`, and T0 succeeds initially. In `acquire()`, trying to acquire the biased lock (`bias`) first (line 9 in Figure 2.10) allows the same thread, here T0, to execute the CS in a loop without interfering with other threads because of the lock monopolization behavior.

Since both T1 and T2 fail to acquire `bias`, they will attempt to acquire a FIFO lock, `fifoH` (line 10). Only T1 will succeed and become the *candidate* for entering the CS after T0; T2 is queued in `fifoH`. The candidate T1 waits on the biased lock for its turn (line 12) but is able to succeed only if T0 releases and does not immediately reacquire the lock. When that happens, T1 becomes the owner of `bias`, and it elects T2 waiting on `fifoH` as the candidate by releasing `fifoH` (line 14).

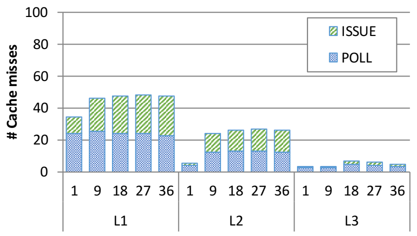
On the other hand, when `acquire_low()` is used (as for T3 and T4), the FIFO behavior is maintained as threads will be queued up in `fifoL` (line 19). `filter` serves as a mechanism to switch between low-priority and high-priority paths when low and high priorities are mixed.



(a) Message rate for 64-byte message.



(b) Execution time breakdown.



(c) Breakdown of cache misses occurred in `MPI_Isend` or `MPI_Irecv` (ISSUE) and in `MPI_Wait` (POLL).

Figure 2.12: Results of using the selective reactivation technique in combination with the P-CLH lock (denoted as P-CLH-USC) and our hierarchical locality-preserving lock (denoted as P-HCLHub-USC). The timing overhead is about 8% on average (harmonic mean) of the total execution time in the timing breakdown analysis.

The low-priority thread runs only when there is not a concurrently running high-priority thread.

Now, we combine our locality-preserving lock with a NUMA-aware lock using the *lock cohorting* technique [33]. The technique uses a hierarchical locking strategy (i.e., two levels of lock, one at each NUMA node and the other at global scope) in order to allow prioritizing ownership passing to threads in the same NUMA node and to make it NUMA-aware with minimal cost.

Since fairness is not needed at the high-priority branch, we replace `fifoH` in Figure 2.10 with this NUMA-aware lock to further improve cache locality. Note that this hierarchical locality-preserving lock results in NUMA-node level unbounded lock monopolization. We implemented this lock using a spin lock for the global scope and a CLH lock for the NUMA node, which is similar to C-BO-MCS described in [33] except that we replace MCS with our existing CLH lock implementation and we do not employ back-off for the spinlock.

Figure 2.12 presents the optimization results for the message rate. Although some performance loss still occurs, we are able to obtain a message rate of 2 million messages per second with 36 threads. This improvement is due to the significant reduction in the number of cache misses across all caches, as shown in Figure 2.12(c), compared with that shown in Figure 2.4(b).

This performance number is the best message rate recorded for this benchmark with this large number of kernel threads. The final result achieves five folds performance improvement compared with the performance of the baseline case.

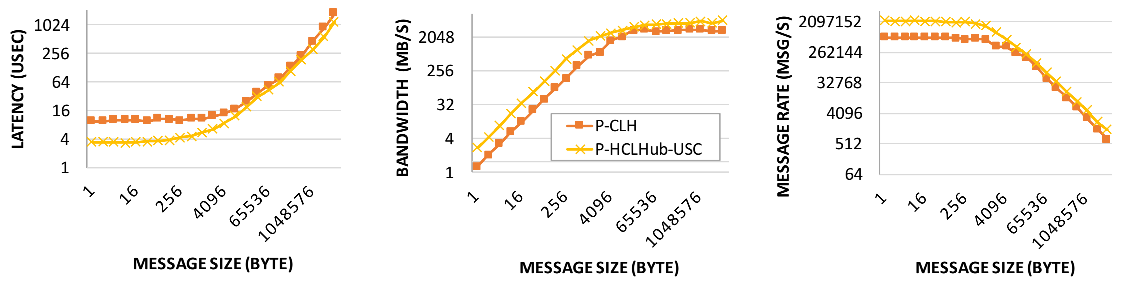
2.4 PERFORMANCE EVALUATION

We performed the experiments of this section in the Stampede cluster [45]. Each node is equipped with a dual-XEON E5 processor, in other words, 16 cores in total. Although the number of cores is less than that of the previous cluster used for running the benchmarks (which is less ideal for our cases), Stampede allows us to experiment with more machine nodes while using the same network interface (an FDR Mellanox device).

We compiled our programs and with the Intel compiler version 15.0.2 using `-O3` optimization. The MPI uses the default configuration (`-O2`). The MXM layer for the MPI implementation was the same as previously described. Unless explicitly mentioned, all our tests were performed with 16 threads per node, where a thread is bound to a machine core (using `OMP_PROC_BIND=close; OMP_PLACES=cores`).

In each of these experiment, we compare two MPI implementations: one using a priority-based FIFO lock (CLHP_LOCK) and one with with all of our optimizations (SCLHP_LOCK+WDTS). We focus on the relative performance and shows the speedup as the improvement over the baseline CLHP_LOCK.

2.4.1 Communication Benchmarks



(a) Latency (the lower the better)

(b) Bandwidth (the higher the better)

(c) Message Rate (the higher the better)

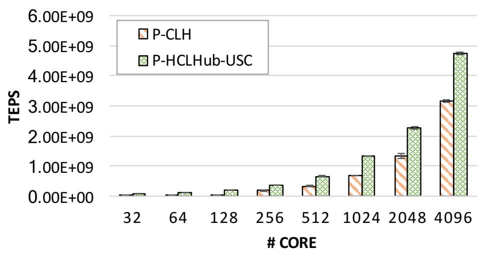
Figure 2.13: Performance results in terms of latency, bandwidth and message rate for 16 threads with variable message size. Secondary y-axis shows the speedup.

Figure 2.13 shows our results for the communication benchmarks that we have used in previous sections in the production supercomputer. This performance is consistent with previous results. Our method outperforms the baseline for all tested cases. The improvement of our combined method ranges from 40% to 3.5× depending on the message size.

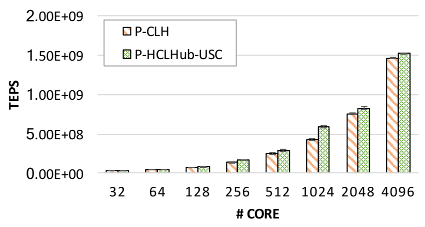
2.4.2 Graph500

Graph500 [46] is a communication kernel that generates a large-scale graph, assigns to each MPI process a fixed set of vertices, and cooperatively traverses the graph in a breadth-first search until all vertices are visited. The kernel represents irregular access patterns and fine-grained communication, which is frequently used to evaluate the communication layer of programming models and runtime system.

Our reference implementation is obtained from the hybrid approach implementation described in [47]. We improved this implementation by converting nonblocking calls to blocking calls and dedicating a subset of threads (half in this experiment) to perform message receiving. We assign the odd threads as sender and event threads as receiver so that the locality of data is better. We find that this implementation outperforms the reference implementation; and since blocking calls are used, we can evaluate our methods better.



(a) Weak scaling for problem size scale 24 per node (32 at 256 node), 16 threads per node, in terms of traversed edges per second (TEPS - the higher the better)



(b) Break down in timing for communication, computation and OMP thread synchronization (Sync) with increasing number of cores with and without communication-aware technique.

Figure 2.14: Graph500 performance from reported harmonic mean over 16 runs. The `Speedup` shows relative performance between them.

Our performance results are shown in Figure 2.14. The weak scaling shows our optimization consistently outperforms the baseline. This is due to two reasons.

First, the miniapp is communication-bounded as shown in Figure 2.14 (b). SCLHP_-LOCK+WDTS improves the communication which lower their percentage with respect to computation and synchronization in all cases.

Second, our methods execute MPI more efficiently which reduces the number of cycles a thread spent inside MPI as shown in Figure 2.14 (c). The variation in cycle of CLHP_LOCK in combination with the increased in number of polls in some cases indicates synchronization conflicts and wasted CS inside MPI, which is greatly reduced using our implementation. The cycles spent inside MPI per message clearly reflects the trend of overall speedup (up to 3.8×

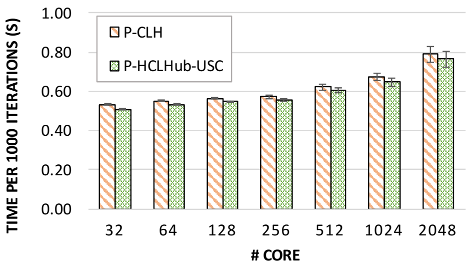
depending on the problem size) that we achieved in Figure 2.14 (a).

2.4.3 HPCCG

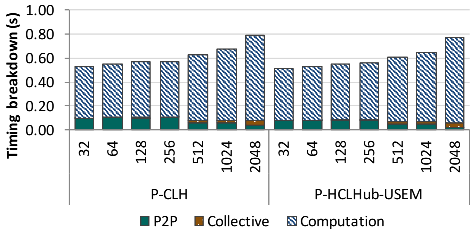
HPCCG is a miniapp from the Mantevo benchmark suite [48]. The miniapp represents a close approximation to a finite-volume application. The communication pattern is irregular, mainly due to several of Sparse Matrix Vector Multiplication steps generated by the application. The problem size is determined by the size of the matrix, in this case generated by the number of rows per process. The communication is performed prior to the local computation using mainly point-to-point MPI calls.

The benchmark suite also provides three different implementations: MPI-only, MPI+OpenMP, and OpenMP. The MPI+OpenMP implementation, however, is a MPI_THREAD_-SINGLE application where OpenMP is used only for parallel loops. We refer to this MPI+OpenMP as the reference implementation. Our strategy for the hybrid implementation is to further subdivide the matrix into smaller domains and assign those smaller domains to each thread. Thus, each thread has to perform both communication and computation.

Communication between threads in a node is done via shared memory for collectives and shared states with appropriate synchronization. Where this is not possible, we used MPI by sending and receiving the data with itself using appropriate tags.



(a) Weak scaling execution time in terms of time per 1K iterations.



(b) Time percentage of point-to-point (P2P) vs. Allreduce operation with respect to overall runtime

Figure 2.15: Performance for HPCCG with 128K matrix rows per node, 16 threads per node.

The performance results are shown in Figure 2.15. With our optimizations we are able to outperform the baseline method from 3-5% as shown in Figure 2.15 (a). However, the performance improvement is small and only statistically significant upto 64 nodes due to the dominant of computation and allreduce operation which is not improved by our algorithms (as shown in Figure 2.15 (b).

2.5 RELATED WORKS

Some early work on supporting the interoperability between threads and MPI, such as MiMPI [49] and MPICH-MT [50], focused only on thread safety issues, since multicore machines did not exist at that time. Another approach is to implement MPI processes as threads [18, 51, 52, 53, 54]. While this approach can bring performance benefits for on-node communication by exploiting efficient data sharing between threads, it requires completely new implementations of both the MPI runtime and shared-memory programming model runtime. It also needs compiler support to privatize global variables for each thread because MPI processes, which are implemented as threads, have to own separate memory space for global variables.

The issue of granularity and arbitration in supporting thread safety has been studied before. For example, Dózsa et al. [32] and Balaji et al. [31] studied the replacement of the MPI coarse-grained lock with fine-grained locks and implemented parallel receive queues using these locks. The implementation proved to be complex and error-prone, however, and thus was not completed. On the other hand, thread arbitration was studied in detail first in [36, 40] and showed significant improvements. In this paper, we have generalized the previous techniques and improved upon the implementation.

Off-loading MPI communication or network polling to dedicated cores/threads is another theme of research. MPICH-Madeleine [55] is one of the early MPICH-based implementation using this approach for supporting threads; it also creates a separate thread for each non-blocking call, which is different and costlier than our design. The approach is more often used when MPI communication is integrated into a light-weight threading runtime such as Habanero-C [56], or Qthreads [57]. Liu et al. [58] demonstrates a general approach to incorporating user-level threading and MPI, giving different methods for network polling. We share the polling mechanism. However, these designs are based on top-down solutions: adding extra layers atop of MPI and therefore having higher latency.

It is worth mentioning that multi-threaded communication can be a solution for heavily communication-bound applications on multi-core clusters. In this approach, multiple threads or cores are cooperating to execute communication related codes. USFMPI [59], MT-MPI [60], and pioman [61] are a few that follow this direction. The technique is orthogonal to our work and is useful when more than one communication server is required to cope with a higher message injection rate. One must also watch out for performance degradation when there are too many concurrent messages in the current NIC architecture, as pointed out by Luo et al. in [62]. Our packet pool design has already taken that into account.

Communication-aware techniques have been proposed in other related contexts. AMPI [18] can selectively schedule only threads whose MPI requests have completed, since it is built on Charm++ [63]. Charm++ provides a message-driven execution model, in which the arrival of messages triggers the execution of appropriate *chares*, the Charm++ word for a task. A similar technique was studied in [64], where a CPU core was used for progress and to partly control the continuation of OS-level threads by converting blocking calls to nonblocking calls. Further improvement can be made by a more tightly coupled design between the thread scheduler and the network interface [65]. In these works, however, executions require a centralized entity to control the executions of other entities and thus can result in wasting a CPU core for the dedicated scheduler.

2.6 SUMMARY AND IMPACT

In this work, we tackled the problem of thread arbitration and synchronization in the context of MPI, and we proposed thread synchronization techniques to improve the communication performance in multithreaded communication scenarios using `MPI_THREAD_MULTIPLE`.

Our techniques reduce the wasted time in the critical section while preserving data locality. Our method adopts a synchronization counter-based selective wakeup mechanism to reactivate waiting threads. It relies on electing and assigning at most one waiting thread to drive a communication context for improved data locality. Furthermore, active threads are prioritized and synchronized by using a locality-preserving lock that is hierarchical and exploits unbounded bias for high throughput.

Our method does not count on an additional dedicated communication server but is incorporated into the implementation in a decentralized manner, which produces a scalable runtime system. We implemented our techniques in a production MPI implementation, MPICH. Experimental results on multicore clusters show significant improvement in synthetic microbenchmarks and two MPI+OpenMP applications.

Since published in [66], the techniques in this work have been incorporated into MPICH version 3.3 [67]; OpenMPI has also moved towards a similar algorithm for handling threads [68].

CHAPTER 3: COMMUNICATION IN GRAPH ANALYTICS FRAMEWORKS

The performance of graph analytics applications on large-scale clusters is usually limited by communication. These applications are built using a variety of frameworks [69, 70, 71, 72, 73, 74, 75, 76, 77, 78] that are in turn implemented on top of TCP or MPI. MPI can provide better performance than TCP, but it is not an ideal communication interface for graph analytics. Indeed, applications in this domain are quite different in their behavior from the scientific computing applications that MPI was designed for, for the following reasons:

- Graph analytics applications are less compute-intensive than typical scientific applications; in fact, clusters are used in graph analytics for their large memory rather than their computational capability. This makes it difficult to overlap communication with computation to reduce the performance impact of communication.
- Graphs that arise in traditional HPC applications are often uniform-degree graphs, and their total size increases polynomially in their average diameter. In contrast, graphs of interest in graph analytics are power-law graphs, and their total size is exponential in the average diameter [79]. For many problems, it is necessary to use different parallel algorithms for these two classes of graphs [80].
- Most traditional HPC applications are *topology-driven* in which identical operations are performed on each node of a graph. In contrast, efficient graph analytics algorithms are *data-driven algorithms* in which computations are performed only at certain *active* nodes in the graph. Nodes become active in data-dependent, statically unpredictable ways, so the patterns of control-flow and data accesses are much more irregular [81].

Because of these factors, an efficient communication system is even more critical for good end-to-end performance of graph analytics applications than it is for traditional HPC applications.

Nevertheless, the graph analytic frameworks (Abelian and Gemini) studied in this chapter exposes that, MPI semantics are not the best match to the needs of graph analytics computations. Some MPI features, such as its strict message ordering requirements or its support for wild-cards, are known to be impediments to high message rates, especially with many concurrent communications [82].

In addition, MPI does not efficiently support receiving messages of unknown size: one either needs to allocate buffers of maximum message size or add a probing call (`MPI_PROBE` or `MPI_IPROBE`) to find out message size. MPI implementation also originally does not provide support for message-driven scheduling of threads; this is usually implemented by a polling agent that sits on top of MPI, separate from the polling done by the progress engine of MPI. Finally, several MPI features such as data-types are not needed by the applications, but add overheads in the critical path.

3.1 OVERVIEW OF GRAPH ANALYTICS FRAMEWORKS

The Gemini [75] and Abelian [83] systems support *vertex programs*: some of the nodes in the graph are initially *active*, and applying an *operator* to an active node makes it inactive and may make some of its neighbors active. An operator can only access the labels of the active node and its immediate neighbors. A *push-style* operator reads the active node's label and writes its neighbors' labels, and a *pull-style* operator reads its neighbors' labels and writes the active node's label. Computation terminates when all nodes are quiescent.

On distributed-memory clusters, the graph is partitioned among hosts using one of many partitioning policies. Gemini only supports the edge-cut partitioning strategy whereas Abelian supports general partitioning strategies including edge-cuts and vertex-cuts. For a edge-partitioned graph, if an edge (u, v) is assigned to a host, the host creates *proxies* for nodes u and v and connects them with an edge. Since the edges connected to a given node in the graph may be partitioned among several hosts, it is possible for a node to have many proxies in the partitioned graph. One of these proxies is designated the *master* proxy while the rest are designated as *mirror* proxies. The master proxy is responsible for the canonical value of the vertex for which it is a proxy.

Since the edges incident to a node may be assigned to different hosts, a node may have multiple proxies. Since the proxies for a given graph node may be read and written by different hosts, we need a synchronization strategy to coordinate the reads and writes. One approach is to use distributed shared-memory (DSM) [76] but the overheads of this approach are substantial. Instead, Abelian and Gemini use the Bulk-Synchronous Parallel (BSP) model for synchronization [84]. The program is executed in rounds, and each round consists of computation followed by communication.

During the computation phase, each host applies the operators to active nodes in its partition. The communication phase is used to synchronize the labels of all proxies, and it can be composed from two patterns. The first, *reduce*, has all mirror proxies communicate their values to the master proxy, where the master proxy combines them into a canonical value. The second, *broadcast*, has the master proxy communicate the canonical value to all mirror proxies. Depending on the partitioning policy used as well as the operator (push or pull), reduce, broadcast, or both are necessary to synchronize the required values. For

```
1  struct NodeData { // data on each node
2    float rank;
3    float residual;
4    unsigned int nout;
5  };
6
7  struct pageRank {
8    Graph* graph;
9    void operator()(GNode src, Abelian::Context& ctx) {
10     auto& sdata = graph->getData(src);
11     auto residual_old = sdata.residual.exchange(0.0);
12     sdata.rank += residual_old; //apply residual to self
13     auto delta = residual_old*alpha/sdata.nout;
14     for(auto nbr : graph->getNeighbor(src)){
15       GNode dst = graph->getEdgeDst(nbr);
16       auto& dResidual = graph->getData(dst).residual;
17       dResidual += delta; // update residual of dest
18       if(dResidual > tolerance)
19         ctx.push(dst); // push to the work-list
20     }
21   }
22 };
23 Abelian::for_each(graph, pageRank{graph});
```

Figure 3.1: PageRank: push-style, data-driven version

example, if a graph node do not have incoming edges or they are assigned to a single partition, no *reduce* should be required.

The major difference between Abelian and Gemini is that the Abelian runtime is partition-aware. It minimizes the communication volume by choosing reduce, broadcast, or both, based on the partitioning policy. It also minimizes the communication meta-data while synchronizing only the updated labels, thereby further reducing communication volume. Gemini supports only edge-cut partitioning, the programming style is similar to Abelian with operator definitions and explicit communication calls; Abelian contains a compiler that injects the communication codes [83].

Figure 3.1 shows an Abelian program for a push-style, data-driven computation of pagerank [85] written by a programmer. The **Abelian::for-each** in line 23 constructs the work-list of active nodes, populating it initially with all the nodes in the graph, and then iterates over it until the work-list is empty. The work-list is passed to the operator in the context `ctx`. The operator computes the update to the pagerank of the active node (delta), and then pushes this update to all the neighbors of the active node. If the residual at a neighbor exceeds some user-specified threshold, that neighbor becomes active and is pushed to the work-list. Note that code has no explicitly parallel constructs.

3.2 MPI IMPLEMENTATIONS

To understand how the communication pattern is implemented, it is necessary to consider the in-memory representation of graphs in the Abelian system. On each host, the master nodes are stored contiguously, followed by mirror nodes. The data for each graph node is usually a `struct` with several labels or fields and this data is stored as an array of structs (AoS).

Not all nodes are active at the same time, and not all labels need to be communicated. Thus, the data to be communicated from one host to another is not contiguous, neither on the send side, nor on the receive side. The layouts of communicated data in sender memory and receiver memory are not identical. Furthermore, what data is communicated changes at each round, and is data-dependent. Sending each entry in an individual message is not practical, nor is it feasible to use `MPI_REDUCE`.

In general, each host may need to communicate with each other host. Each send to another host is preceded by a *gather* operation that stores in a contiguous buffer the data to be sent to that host; each receive from another host is followed by a *scatter* operation that retrieves data from the contiguous receive buffer and updates the relevant entries.

The update involves copying the data if the communication is a broadcast, or applying a reduce operation, if the operation is a reduce. The gather and scatter patterns may involve different sets of hosts at each round. Thus, both cases result in a *gather-communicate-scatter* pattern of communication. Note that the same communication pattern holds for other distributed memory graph frameworks.

If a host needs to communicate values to m other hosts, it is necessary to perform m gather operations. These gather operations are independent and in principle can be performed in parallel. Scatter operations can also be parallelized, taking care to avoid races for the reduce updates.

The communication pattern can be implemented in MPI using either one-sided or two-sided methods. We next discuss these implementations and their limitations.

3.2.1 MPI Two-sided Implementation

The gather-communicate-scatter pattern is implemented in Abelian as follows. One thread on each host is dedicated to communication with other hosts. The other threads perform computation during the local computation phase, but they also participate in communication during the communication phase; to keep the terminology simple, we refer to them nevertheless as *compute threads.*The compute threads perform gathers into buffers in parallel. Completed buffers are enqueued to the send queue of the communication thread. Once gathers are complete, the compute threads switch to performing scatters in parallel to process messages received from other hosts. The messages from other hosts are processed in an arbitrary order as they arrive.

The dedicated communication thread interleaves sending and receiving. It checks to see if there are buffers that need to be sent, and if so, it pushes them out into the network. It also polls the network for incoming messages, and enqueues them into a receive queue to be processed by the compute threads. To maximize throughput, no blocking operations are used. This design is intended to reduce the overhead of communication by parallelizing gathers and scatters, and overlapping them with communication, and minimize latency and overheads introduced by the concurrent access to MPI. The `MPI_THREAD_FUNNELED` mode is used, and MPI commands are only issued from the dedicated communication thread. There are two problems with this design.

First, the lack of back pressure on producers when data is produced faster than consumed. This is especially problematic with MPI's eager protocol, as it may lead to the exhaustion of MPI data buffers. The slow consumption can occur either at the network interface due to packet injection rate limits on many networks, resulting in buffer exhaustion on the sending side, or at the consumer, resulting in buffer exhaustion on the receiving side.

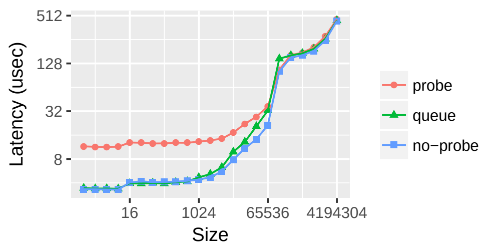
The lack of back pressure also increases buffer consumption at the application level, because of the all-to-all communication pattern. Without a solution, MPI implementation may hang or crash due to out-of-resources exception.

To resolve the lack of back pressure, the Abelian solution is to add a buffered network layer which works as follows. For sending messages (maybe different datatypes or fields), the system buffers small items (those less than the eager-send limit) until either the oldest buffered message times out or the buffer size exceeds the eager send limit.

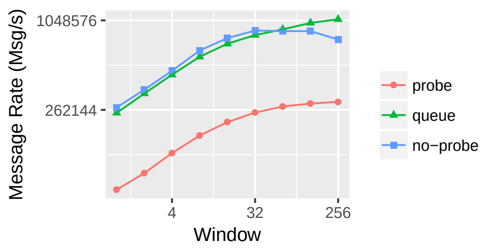
This solution reduces buffer consumption of MPI, while capping latency. Custom buffering is done via thread-safe multi-producer, single consumer queues, the communication thread pops from this queue and only interacts with MPI serially, which minimizes the thread synchronization overheads and controls the memory usage of MPI. This solution however, adds extra overheads over the critical path of the application.

The second issue happens at the receiver. `MPI_IRECV` cannot be used directly, since the communication is irregular and dynamic (*e.g.*, there is no prior information about the incoming message size). Abelian makes use of `MPI_IPROBE`[1] with MPI wildcards to handle

[1]We did not try `MPI_IMPROBE / MPI_MRECV` since all communication is done by one thread and, in our experiments with both MVAPICH2 and IntelMPI, `MPI_IMPROBE / MPI_MRECV` is slower and/or hang with various benchmarks.



(a) Latency benchmark with increasing data size



(b) Message Rate benchmark with increasing message window size for 64-byte message

Figure 3.2: Performance comparison of MPI_SEND/RECV (no-probe), MPI_PROBE (probe) and LCI (queue) using an adapted OSU latency and message rate test [43] in the Stampede2 cluster.

incoming data.

The `MPI_STATUS` returned by this function provides the information to start receiving the message, such as the size of the buffer, the source and tag of the message. Subsequently, a `MPI_IRECV` is called to continue with the pending communication. The communication thread uses `MPI_TEST` calls to ensure forward progress. It reclaims buffer space as communications, both sends and receives, are complete.

This pattern of probing for messages is also problematic for MPI. Firstly, it doubles the amount of message-matching, one in `MPI_IPROBE` and one in `MPI_IRECV`. Secondly, it is inefficient for multi-threaded usecase [86, 87]. Thirdly, all messages are unexpected and requires double buffering in the runtime and in the application. Both Abelian and Gemini however use this pattern. Figure 3.2 shows the inefficiency of this pattern compared to the regular MPI pattern.

3.2.2 MPI One-sided Implementation

MPI3 introduces one-sided communication via remote memory access(RMA), which allows a direct path to RDMA capability of the device and lower the communication overheads of matching data via matching queues. Abelian implemented a communication layer using MPI3 RMA. The goal of this MPI RMA implementation is to lower the communication overheads and other down-side of two-sided communication mentioned above.

Typical MPI RMA implementations [88] are single-threaded, store the entire graph in a RMA window, and access the graph during computation via MPI RMA calls, thus limiting computation efficiency and the opportunity for communication aggregation. In contrast, Abelian creates RMA windows only for receiving aggregated messages during communication so that the computation accesses only the local graph.

The main challenge in achieving this is in determining the receive buffer sizes since they need to be pre-allocated. As mentioned earlier, in graph analytics application, the data communicated in each round of communication varies widely, even between the same pair of hosts, however an upper bound can be computed assuming all nodes are active.

In the MPI-RMA communication layer, compute thread performs RMA operations. To send messages, a host will start an access epoch on its RMA window (`MPI_WIN_START`). For each destination, a parallel gather by all compute threads is performed. This prepares the send buffer (source data), which is then written using `MPI_PUT` to the remote memory or buffer of that destination in the host's window.

Finally, after the `MPI_PUT` is initiated for all remote destinations, the host will complete the access epoch on its window (`MPI_WIN_COMPLETE`). A host exposes its receive buffers (in different windows) at the end of a round with calls to `MPI_WIN_POST` and checks for the completion of the remote access to its receive buffers with calls to `MPI_WIN_WAIT` after completing the access epoch on its window. When the `MPI_WIN_WAIT` for a remote source's RMA window returns, the host performs a parallel scatter to process the (local) received buffer and then posts a new exposure epoch on the source's window using `MPI_POST`.

The dedicated communication thread in the one-sided MPI communication layer does not interact with the computation threads. However, the dedicated communication thread continuously polls the network (`MPI_IPROBE`) to ensure forward progress [89] for the MPI RMA operations. Since both compute threads and the dedicated communication thread are issuing MPI commands, this layer uses `MPI_THREAD_MULTIPLE`.

The main issue with the MPI RMA implementations is the amount of allocated memory for the windows. Since the amount of data is unknown (due to unknown active vertices at each round), Abelian conservatively allocates a maximum amount of possible buffering in the MPI window. This reduces the locality of the overall communication compared to the two-sided version, while communication buffers can be reused.

3.3 LOW-LEVEL COMMUNICATION INTERFACE

Based on the study of Abelian and Gemini implementation using MPI layer, we describe the Light communication interface (LCI) design and implementation which directly maps communication calls to the lower level network interface. LCI not only reduces the latency of messages when compared to two-sided or one-sided MPI implementations, but also dynamically manages memory requirements, thereby reducing memory usage compared to the one-sided MPI implementation.

We present in this section only the design of **Queue**, an LCI interface specializing to support Abelian and similar irregular communication patterns. The goal of Queue is to avoid the short-comings of MPI implementations described above.

- *LCI avoids fatal failures due to insufficient network resources.* This is done by allowing the upper layer to retry the operation on such events. The MPI standard does not require implementations to handle resource exhaustion errors and in current MPI implementations the program crashes when these happen. This problem has to be mitigated by an additional buffered layer as discussed before.
- *LCI supports multi-threading efficiently with a communication server.* The interaction between the server and the compute thread is limited to a single flag. This is not possible in MPI; a `MPI_TEST` leads to an expensive network poll, thus leading to additional operations for checking request completion.
- *LCI is closer to the network interface.* This prevents any buffering and duplicated functionality due to the complexity of the MPI implementation (e.g. there is no tag-matching or ordering enforcement in the LCI interface).

Communication in LCI involves the following two steps.

Communication Initiation: Communication is started by obtaining resources for sending data or checking if there is an incoming packet to process. When successful, the call returns a request handle which contains a record of the communication and the resources corresponding to the communication. In comparison to an MPI non-blocking function (such as `MPI_ISEND`), our initiation can fail if there are no available resources (for sender) or there is no pending communication (for receiver). However the failure is not fatal and simply means the user should retry at a later time. The two functions for initiation of send and receive are **`SEND-ENQ`** and **`RECV-DEQ`** respectively.

Communication Completion After initiation is successful, the communication is now in progress. The progress is implicit and typically ensured by a communication server. When the communication is finished, a boolean flag is set. In comparison to MPI functions such as `MPI_TEST` or `MPI_WAIT`, our mechanism is more lightweight: there is no need for a function call; the user maintains a list of requests and checks the status flag fields.

To implement Queue, we make use of some abstractions for interacting with the underlying network APIs. We present here a simplified version of this list of functions:

- **lc_send(p)**: submit a command to the network which transfers a limited-size packet structure (p) enclosing a header with some information such as a rank and tag of the

Algorithm 3.1 SEND-ENQ operation (executed by thread)

```
1: $P$: a global concurrent packet pool.
2:
3: procedure SEND-ENQ($b, s, h, t$)                    ▷ : buffer, size, rank, tag
4:     $p$ = packetAlloc($P, s, h, t$)
5:     if $p$ then
6:         $r$ = makeRequest($p$)
7:         if $s$ is small then
8:             copy($p.b \leftarrow b$)
9:             $p.type \leftarrow$ EGR
10:            lc_send($p$)
11:            $r.status \leftarrow$ DONE
12:        else
13:            $r.status \leftarrow$ PENDING
14:            $p.src \leftarrow b$
15:            $p.type \leftarrow$ RTS
16:            lc_send($p$)
17:        end if
18:        return $r$
19:    end if
20:    return NULL
21: end procedure
```

destination, the type of the packet, and some data. Every host has to maintain a fixed number of buffers for receiving these packets.

- **lc_put(**p**,** $src \rightarrow dst$**)**: submit a command to the network which transfers data from a source buffer (src) to a target buffer (dst), identified by a host and key for address translation enclosed in the packet p.

- **lc_progress()**: ensure progress of the communication such as flushing outgoing data and peeking for an incoming packet. If a packet is received, from any host, the function returns it to the caller.

`lc_send` and `lc_put` are non-blocking and are typically very short. They can be executed by both communication and computation threads. `lc_progress` can take longer since it typically requires draining the network driver by executing the network progressing functions. Hence, it is only executed by the communication thread.

`lc_send` is provided by most network interface APIs and is typically used for short messages in an eager protocol. `lc_put` can be implemented directly by the hardware if the

Algorithm 3.2 RECV-DEQ interface (executed by thread)

```
1:  $Q$: a global concurrent queue.
2:  $P$: a global concurrent packet pool.
3:
4:  procedure RECV-DEQ($*b, *s, *h, *t$)
5:                                              ▷ : pointer to buffer, size, rank, tag
6:      $p$ = dequeue($Q$)
7:      if $!p$ then                                                ▷ $Q$ is empty
8:          return NULL
9:      end if
10:     $r \leftarrow$ makeRequest($p$)
11:     $(*s, *h, *t) \leftarrow p.header$
12:     $*b \leftarrow$ allocate($*s$)
13:     if $p$ is EGR then
14:         copy($*b \leftarrow p.b$)
15:         $r.status \leftarrow$ DONE
16:         packetFree($p$)
17:     else
18:         $p.dst \leftarrow *b$
19:         $r.status \leftarrow$ PENDING
20:         $p.type \leftarrow$ RTR
21:         lc_send($p$)
22:     end if
23:     return $r$;
24: end procedure
```

network interface supports RDMA. In particular, for *psm2*, the native network API of Omni-Path, `lc_put` is implemented by translating target identification to a special tag. This is convenient enough since *psm2* has a rich set of tag-matching interfaces (96 bits can be used for matching purposes).

On the other hand, for *ibverbs* of Infiniband devices, our implementation using reliable connection (RC) is straightforward: both `lc_send` and `lc_put` map directly to `ibv_post_send` calls using `IBV_WR_SEND` and `IBV_WR_RDMA_WRITE` work request respectively [2].

The pseudocode for send and receive with the Queue interface is presented in Algorithms 3.1 and 3.2 for both eager and rendezvous protocols (selected automatically depending on the size of the incoming buffer). A request is a structure for storing the ongoing communication status and ties to a packet for flow control.

The rest of the communication is done by the communication server as presented in

[2]RC is sufficient for our current purpose since we maintain one process per host. One can also emulate RDMA atop other connection types like in [90, 91].

Algorithm 3.3 Network progress (executed by server)

```
1: $Q$: a global concurrent queue.
2: $P$: a global concurrent packet pool.
3:
4: procedure NETWORK-PROGRESS
5:     $p \leftarrow$ lc_progress
6:     if $p.type$ is EGR or RTS then
7:         enqueue($Q, p$)
8:     else if $p.type$ is RTR then
9:         $p.type \leftarrow$ RDMA
10:        lc_put($p$, $p.src \rightarrow p.dst$)
11:    else if $p.type$ is RDMA then
12:        $p.r.status \leftarrow$ DONE
13:        packetFree($P, p$)
14:    end if
15: end procedure
```

Algorithm 3.3. The basic idea is for the communication server to progress the network and execute appropriate callbacks for each packet type. Specifically, packet types are as follows: **EGR** - eager packet for short protocol which includes the data; **RTS, RTR** - ready-to-send and ready-to-receive control packets respectively, which are commonly used for the rendezvous protocol to exchange the buffer addresses; and **RDMA** - packet specifically for the `lc_put` operation.

The algorithms also rely on two variables P and Q which are accessed atomically for supporting thread-safety: *packetAlloc/Free* - to allocate/free a packet; *enqueue/dequeue* - to store/retrieve incoming packets. These operations can be easily implemented with a concurrent pool and a concurrent queue respectively. We implemented the locality-aware packet pool presented in [82] and the fetch-and-add based MPMC queue presented in [92].

The size of the packet pool determines the maximum injection rate, which is typically a small constant times the number of hosts. The *allocator* can be any thread-safe memory manager; in our case, it is Abelian's allocator.

Due to its simple semantics, Queue can maintain a short matching queue at all times. Unlike MPI, ordering semantics are not required and not enforced. Instead, the `SEND-ENQ` returns the first arriving packet. We name this the *first-packet policy.* If needed, the user can ensure completion ordering by draining their pending requests before submitting more packets to the network or by maintaining an ordered list of pending requests and checking them in order. In particular, Abelian's communication thread maintains this order with respect to a specific incoming host.

Table 3.1: Inputs and their key properties.

	clueweb12	**kron30**	**rmat28**
$\lvert V\rvert$	978M	1073M	268M
$\lvert E\rvert$	42,574M	10,791M	4,295M
$\lvert E\rvert/\lvert V\rvert$	44	16	16
max D_{out}	7,447	3.2M	4M
max D_{in}	75M	3.2M	0.3M

The simple *first-packet policy* fits naturally into Abelian's communication layer since incoming data within a communication phase can be processed in any order. Further, since this is designed to match our higher layer, a thread can send a serialized message through `SEND-ENQ` and use `RECV-DEQ` for probing incoming messages.

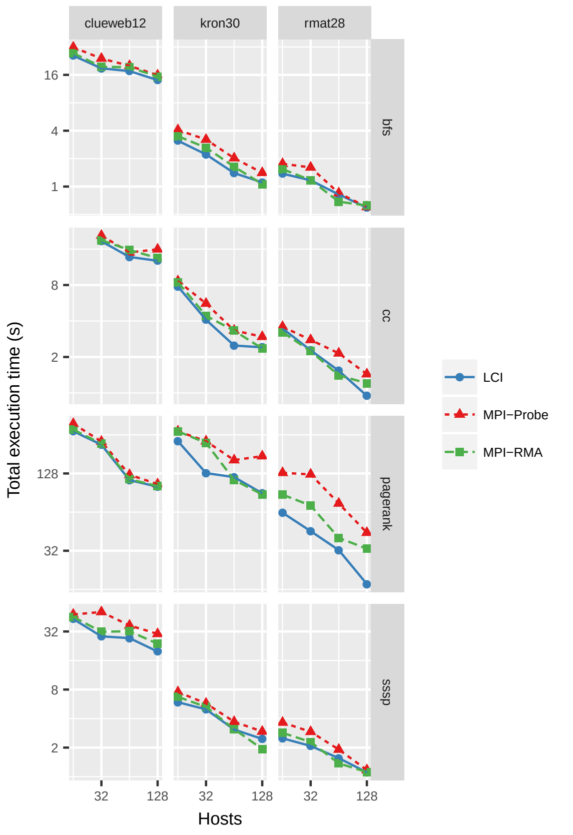
Abelian's communication layer maintains a list of incomplete requests, and can start freeing resources (for sent requests) or deserializing incoming data (for received requests) by simply checking the boolean-type status of each request.

3.4 PERFORMANCE EVALUATION

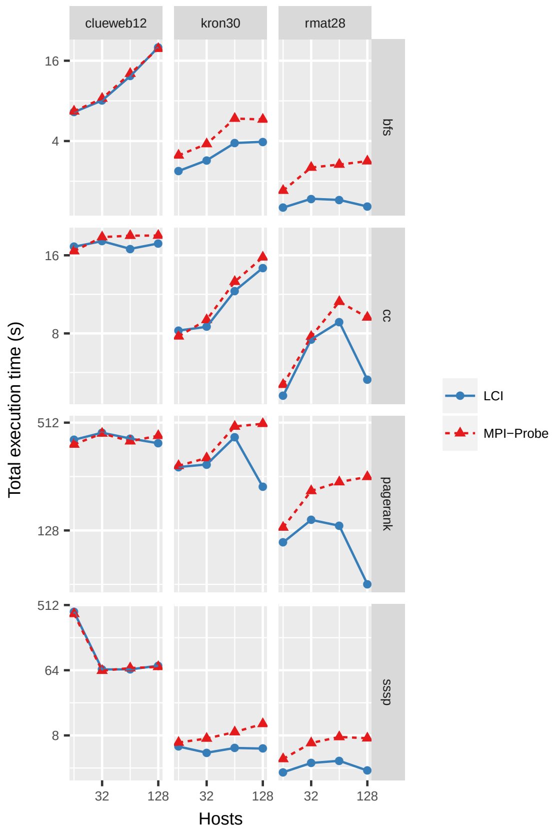
To evaluate the performance of the LCI communication layer on the Abelian and Gemini systems, we used a number of standard graph applications: breadth-first search (**bfs**), connected components (**cc**), single-source shortest path (**sssp**), and pagerank (**pagerank**). Table 3.1 shows the input graphs used in the experiments along with their properties; clueweb12 is one of the largest publicly available web-crawl graphs while rmat28 and kron30 are synthetically generated scale-free graphs. Abelian uses an advanced vertex-cut partitioning policy [93], whereas Gemini uses a simple blocked edge-cut partitioning policy [75] that tries to balance the assigned edges across hosts.

Most of the experiments were done on the Texas Advanced Computing Center's Stampede2 KNL Cluster (Stampede2). We also perform a subset of experiments on the Stampede SandyBridge Cluster (Stampede1). All code is compiled using `gcc` version 7.1.0 and 4.9.3 on Stampede2 and Stampede1 respectively. In each cluster, we selected the default MPI implementation that is available, we also present some results for other MPI implementations. The results are an avarage of 5 runs using one thread per core, excluding graph construction time. All algorithms are run until convergence, except for pagerank which is run up to 100 iterations.

Abelian performance results: Figure 3.3(a) shows the execution time of Abelian programs with the LCI, MPI two-sided (MPI-Probe), and MPI one-sided (MPI-RMA) communication

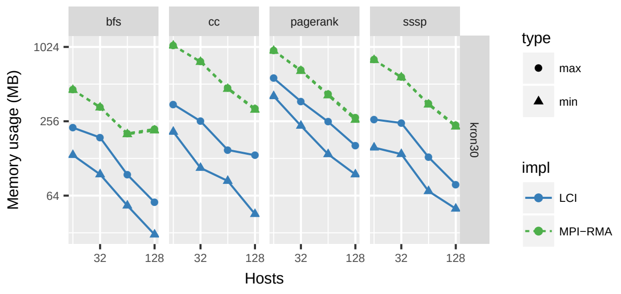


(a) Abelian with LCI, MPI-Probe, and MPI-RMA runtimes.

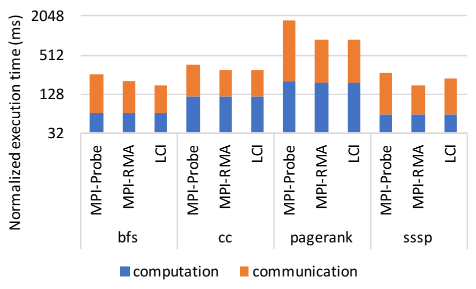


(b) Gemini with LCI and MPI-Probe runtimes.

Figure 3.3: Total execution time on Stampede2 cluster.



(a) Memory usage of communication buffers - maximum and minimum across hosts: Abelian with LCI and MPI-RMA runtimes on Stampede2.



(b) Breakdown of execution time of an iteration on `kron30` at 128 hosts on Stampede2

Figure 3.4: Analysis of the Abelian Performance.

layers. With MPI two-sided, Abelian does not scale well due to the high overheads of multi-threaded communication and the irregular communication patterns that `MPI_PROBE` does not handle well. LCI on the other hand, is able to achieve comparable or better performance than MPI-RMA at various settings. RMA window creation time is excluded in MPI-RMA results, otherwise LCI outperforms MPI-RMA in all cases.

We also observe that the improvement is more significant when the application runs with more iterations where there are more communication rounds like in the case of `pagerank`. The advantage of LCI vs. MPI-RMA is really in the memory usage, which is sometimes reflected in performance. At 128 hosts, LCI achieves a geometric mean speedup of 1.34× over MPI-Probe and 1.08× over MPI-RMA.

To determine the size of the working set of communication buffers, we instrumented the code to count the size of allocation and deallocation of the buffers. The memory usage or footprint of a host is the maximum size of the working set during execution. Figure 3.4(a) shows the maximum and the minimum memory footprints across hosts of LCI compared to MPI-RMA (*this excludes the memory used internally by MPI and only considers the allocated memory by Abelian's code*).

The memory footprint of LCI is much smaller for all applications on all hosts than MPI-RMA. Due to its design, LCI can quickly recycle buffers, thus reducing memory usage and improving locality. Maximum and minimum memory footprints for MPI-RMA are close to each other. The memory usage of MPI-RMA can be up to an order of magnitude higher than that of LCI because MPI-RMA has to preallocate all buffers with a size that is the upper-bound of memory required for communication.

Figure 3.4(b) shows the time spent in computation and non-overlapped communication for `kron30` on 128 hosts. As expected, the changes in performance come from the communication component. In most applications, LCI performs best, or comparable to MPI-RMA. LCI outperforms MPI-Probe since it has lower communication overhead and outperforms MPI-RMA because of the reuse of communication buffers.

Gemini performance results: Figure 3.3(b) presents the total execution time of Gemini with MPI-Probe and LCI. All algorithms are run until convergence. The performance behavior (or difference) is roughly similar to that of the Abelian system. In `kron30` and `rmat28` where communication overheads present a significant fraction of the total communication, we see significant improvement in performance by using LCI. Across all applications at 128 hosts, the geometric mean speedup of LCI over MPI-Probe in communication is 2×, yielding an execution time speedup of 1.64×.

Similarly as Abelian, the performance improvement comes from the better multi-threaded communication. Further, Gemini is using `MPI_PROBE` concurrently, which prevents it to scale

Table 3.2: Total execution time (seconds) for Abelian at 128 hosts using the rmat28 graph on two different clusters.

	Stampede2			**Stampede1**		
	LCI	MPI-Probe	MPI-RMA	LCI	MPI-Probe	MPI-RMA
bfs	0.59	0.60	0.62	0.50	0.52	0.55
cc	0.95	1.44	1.21	1.12	1.15	1.21
pagerank	17.60	44.26	33.21	22.05	23.09	27.65
sssp	1.11	1.17	1.11	1.09	1.12	1.24

Table 3.3: Total execution time (seconds) for Abelian at 128 hosts using kron30 graph with LCI and other MPI implementations on Stampede2. Timing in parentheses are window creation time which are excluded from the other results.

	bfs	cc	pagerank	sssp
LCI	**1.17**	**2.41**	**89.72**	**2.46**
IntelMPI-Probe	1.41	2.95	174.67	2.94
MVAPICH2-Probe	1.40	2.93	177.72	2.82
OpenMPI-Probe	1.33	2.99	171.57	2.82
IntelMPI-RMA (+1.4)	**1.06**	2.36	**87.84**	**1.93**
MVAPICH2-RMA (+1.8)	1.14	**2.29**	93.53	2.13
OpenMPI-RMA (+1.2)	1.21	2.34	93.74	2.25

to large number of threads.

Performance of Abelian on different MPI implementations: One may argue that a better MPI implementation can improve the performance, though we believe the differences between LCI and MPI are fundamental. To verify that via empirical results, we ran some experiments using OpenMPI (commit `f9b157`) and MVAPICH 2.3b (both are latest at the time tested and configured with `psm2`) on Stampede2. The results in Table 3.3 show that LCI remains the winner compared to other MPI implementations. There is no clear winner between different MPI implementations, though IntelMPI-RMA performs best in the majority of cases. LCI is again closest in performance to RMA implementations, and is better if we include time for window creation in the result.

Performance of LCI on other networks: To show that LCI and its performance is portable to other NICs, we ran a subset of experiments on the Stampede1 cluster which is equiped with a Mellanox Infiniband FDR network. We do not focus on this cluster because it has fewer cores on each host (16 compared to 68) and is an older supercomputer. Nevertheless, the results show a similar trend, LCI performs better in all tested cases and closely matches the performance in the Stampede2 cluster as shown in Table 3.2; the exception is that MPI-RMA is actually the slowest. We believe this is because locality of communication is the bottleneck in this system, which has less cache and a slower memory subsystem than

Stampede2.

3.5 RELATED WORKS

Many frameworks for distributed-memory graph analytics have been discussed in the literature [69, 70, 71, 72, 73, 74, 75, 76, 77, 78]. Most of these systems use either MPI or TCP/IP as the underlying communication layer.

Several communication libraries have been developed to provide lower-level support to parallel programming languages and libraries. This includes ARMCI [94], an library developed to support Global Arrays, and GASNet [95], developed to support PGAS languages such as UPC [96]. Neither of these libraries is designed to cope with high thread counts.

Other communication libraries based on the active message (AM) paradigm have been proposed as appropriate for problems with irregular, dynamic communication patterns [97, 98]. The use of AM provides great flexibility but introduces an unnecessary software overhead for many simple data-transfer patterns. Moreover, it is typically prohibited to perform blocking or time-consuming operations on an AM handler. They may also force the use of a CPU proxy for communications targeting GPUs. A possible promising direction would be to upload handlers to the NIC [99] but this raises system management issues that have plagued similar approaches in the past.

UCX [30] and Libfabric [91] are actively under development as generic communication layers; both provide great flexibility but do not optimize for a specific domain. Further, our initial investigation of these libraries does not show good performance with threads.

3.6 SUMMARY AND IMPACT

This section described our works on distributed-memory graph frameworks, an increasingly important area that requires high-performance technologies. We designed LCI layer targeting this pattern, based upon the studies of a state-of-the-art graph analytics system called Abelian, through analyzing and evaluating the performance of two existing Abelian's MPI-based communication layers.

LCI can be used with other graph analytics systems, such as Gemini, another state-of-the-art graph analytics framework with similar patterns of communication. Our experiments show that LCI-based communication system reduces communication time by a factor of up to 2× for a collection of standard graph analytics benchmarks. LCI requires only a few primitive network operations, allowing it to be easily ported to other systems. We have implemented LCI on top of `ibverbs`, `psm2`, and `Libfabric`, which is sufficient for LCI to run on almost all modern platforms.

The development of Abelian and its communication optimization has been released to public at `https://github.com/IntelligentSoftwareSystems/Galois` and [100]. The work has been gaining popularity and has also integrated with heterogeneous architectures such as GPU, through the work [101], which won a best poster award. LCI described in this chapter is the baseline for our development of a new communication library that targets application frameworks.

CHAPTER 4: EFFICIENT SYNCHRONIZATION WITH BIT-VECTORS

The increased heterogeneity of HPC systems, and the increased variability in CPU speed, due to thermal control, have motivated a renewed interest in task-oriented programming models, where *light-weight tasks* (LWT) are dynamically scheduled when their dependencies are satisfied or other events triggered their execution. Examples include Graph analytics like in previous chapter, OpenMP [102], Charm++ [103], Legion [104] and Parsec [105].

When such systems are implemented on top of distributed memory platforms (as is the case for Charm++, Legion or Parsec), or when they interface with message-passing systems (as is needed for OpenMP, Graph analytics), tasks may often be descheduled and rescheduled, due to dependencies on communication events. Thus, it is important to optimize the performance of the wait-signal synchronization: A task yields or blocks, waiting for a signal, using a *wait* call; and the signal is delivered, by another task or by the communication library, using a *signal* call. We raise a question: *what is the minimum cost for this kind of interaction?*

The fast triggering of a computation upon message arrival is, or course, fundamental to the active message paradigm [25]. But active message handlers usually are short, predefined, stateless function that execute in the context of the communication library [26]; their use do not replace the need for fast scheduling of application tasks. Instead, the active message handler is used to interface with the task scheduler, e.g., by enqueuing the signaled task into a ready queue of runnable tasks.

In the context of communication, the performance of a signal is the most critical, since signal is issued by the communication library and can be a sequential bottleneck. A 400 Gb/s network can deliver a 64 byte message in 1.3ns, and future adapters are expected to handle message rates in excess of 100M/sec. The message handler, which includes the signaling logic, will need to execute in few instructions with only a couple of cache misses; even better, it should be simple enough so that it can be implemented in NIC hardware. The sequential bottleneck of handling incoming messages can be alleviated by splitting traffic and handling multiple streams in parallel, but this is a "solution of last resort", since the static partitioning of message streams causes algorithmic inefficiencies.

The performance of wait is less critical since the wait is executed by a running task, and the cost is amortized with a high enough number of concurrent tasks. Both wait and signal however are equally important for synchronization between tasks, but fairness of their execution is usually not a major consideration.

In a parallel loop construct, tasks can executed in any order; thread libraries such as Cilk mostly use a LIFO scheduling policy [106, 107]. Similarly, support for priorities is also not a major concern. OpenMP provides a `priority` modifier for tasks, but this is only a hint that can be ignored. Some systems require support for high-priority control messages as distinct from low priority data messages. Since tasks are usually non-preemptable, priorities are better handled by having separate task pools for throughput and latency oriented multitasking.

Our goal in this chapter is to develop a scheduler for LWTs, so that a signal operation be extremely fast, without suffering a significant deterioration in the performance of other primitives. We formulate the problem of signal-wait with respect to concurrent communication, then develop a solution to efficiently tackle such issue.

4.1 THE RELAXED PRODUCER-CONSUMER PROBLEM

The most frequent synchronization pattern in parallel computing is producer-consumer synchronization: The producer fills a buffer, marks it full and does not fill it again until it is marked empty; the consumer reads the buffer, marks it empty, and does not consume it again until it is marked full. When producer and consumer are on different nodes, the protocol is mediated by the communication library that acts on behalf of a remote producer or a remote consumer – depending on the type of communication used.

For parallel computing and distributed-memory synchronization, it is often the case that only one-way synchronization is required for each buffer transfer as each producer is also a consumer and the reverse communication completes the handshake. That is, the consumer waits for the producer, but the producer can go ahead placing the data without checking that the buffer is empty. This type of synchronization is typical of event-driven code, where a thread is scheduled when its data is available. The producer of the data knows in advance who is the consumer of such data, or where produced data should be deposited via initialization or via a hand-shake that happens before.

We name this a Relaxed version of Single-Item Producer-Consumer (RSPC) to distinguish from the full protocol where both producer and consumer wait for each other. The generic code for such synchronization is illustrated in Listing 4.1.

Listing 4.1: Producer-Consumer synchronization

```
Producer(s)
  PlaceItem(s.buffer)
  SetFlag(s.flag)
  Notify(s.consumer)

Consumer(s)
  Wait-until-set (s.flag)
  TakeItem(s.buffer)
```

Listing 4.2: Condition Variable solution for RSPC

```
Producer(s)
  PlaceItem(s.buffer)
  Lock(s.mutex)
  SetFlag(s.flag)
  CondSignal(s.cond)
  UnLock(s.mutex)

Consumer(s)
  Lock(s.mutex)
  while (!s.flag) do
    CondWait(s.cond, s.mutex)
  UnLock(s.mutex)
  TakeItem(s.buffer)
```

Listing 4.3: Binary Semaphore or Full/Empty bit for RSPC

```
Producer(s)
  PlaceItem(s.buffer)
  V(s.full)

Consumer(s)
  P(s.full)
  TakeItem(s.buffer)
```

To implement, one can use a polling approach for `Wait-until-set` by busy waiting for the flag, thus `Notify` can be a no-op. This is however often undesirable and unscalable since computing cycles are wasted. A better approach is for the consumer to be descheduled, and later be enabled by the producer. The method is often implemented using Condition Variable synchronization object as in Listing 4.2. Mutual exclusion is required since otherwise the producer could raise the flag right after the flag is checked by the consumer and before the consumer invokes wait.

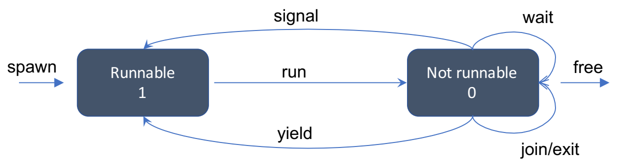


Figure 4.1: State diagram of an LWT and its associated bit value.

Another alternative solution is to use a binary Semaphore such as in Listing 4.3. Semaphore `full` indicates the flag is set and item is ready to be consumed; `V(full)` set counter to 1 and signal any waiting thread; `P(full)` check the counter, if it is 0 then the thread has to wait. [1] The mutual exclusion is not explicit in the code listing but it is still needed; it is hidden in the implementation of the Semaphore.

4.2 THREAD SCHEDULING USING BIT-VECTORS

Like other task schedulers, FULT is maintaining a separate list of runnable tasks at each worker; work-stealing is used to load-balance across workers. The major difference between FULT and other task schedulers is the structure used to maintain this list. We are using for that purpose a bit-vector, where each bit represents the following:

- The index of the bit represents a unique LWT which was assigned this index at spawn time.
- If the value of the bit is 0, the LWT *is not* runnable; it might be running or blocked.
- If the value of the bit is 1, the LWT *might be* runnable.

The use of a bit-vector reduces significantly the overhead of changing the status of an LWT, by marking it runnable (signal) or making it not runnable (when it is scheduled or blocked); it could increase the cost of scheduling a task or of work stealing.

The bit-vector provides an approximate version of the state of an LWT. The actual state of the LWT is found when the scheduler checks the data structure associate with the LWT. This approach is correct and yields good performance if false positives (LWTs marked one but not runnable) are rare.

[1]Since this is a relaxed problem, we do not need an addition Semaphore to make sure the buffer is to empty before the new item is placed.

4.2.1 The Baseline Algorithm

We now describe the baseline algorithm for scheduling using a bit-vector. The baseline version assumes a limited number of concurrent LWTs. In other words, we assume that the time to traverse the entire bit-vector is low. In practice, with a typical 64-byte cache line size, a worker can traverse up to 512 concurrent LWTs with a single cache miss. The traversal can be executed with few instructions, using leading zero count vector intrinsic. We later show how to support larger LWT counts.

We first also assume that an LWT can only perform the following operations:

- `Spawn`: spawn a new LWT on a specific worker.
- `Yield`: An LWT is suspended temporarily but remains runnable.

Algorithm 4.1 Thread scheduler using bit-vector

```
procedure SimpleScheduler($\omega, V$: corresponding worker and its bit-vector)
    while !$\omega$.stop do
        for word in $V$ do
            if word $\neq 0$ then
                localWord := 0
                AtomicExchg(word, localWord)
                while localWord $\neq 0$ do
                    b := Leadingbit(localWord)
                    localWord := ClearBit(localWord, b)
                    t := GetThread($\omega$, b)
                    ContextSwap($\omega$, t)
                end while
            end if
        end for
    end while
end procedure
```

Algorithm 4.2 `Spawn` called by an LWT

```
procedure Spawn($f$, $\omega$)        ▷ function for this LWT, worker that is spawned to.
    b := GetBit($\omega$)
    t := GetThread($\omega$, b)
    t.func := f
    SetBit($\omega$, b)
    return t
end procedure
```

Algorithm 4.3 `Yield`(ω, f): called by an LWT

```
procedure YIELD($f$, $\omega$)          ▷ function for this LWT, worker that is spawned to
    SetBit($\omega$, t.b)
    ContextSwap($t$, $\omega$)
    return t
end procedure
```

The scheduler works as follows. A worker first looks for an LWT that might be runnable by finding one bits in the bit-vector. Then it looks up in another array using the index of the bit to find the LWT descriptor. Once a runnable LWT is found, it is executed by the worker.

The scheduler marks eagerly LWTs that are runnable, so as to make sure they are considered by the scheduler. On the other hand, it can zero the LWT bit lazily, since information on the status of the LWT can also be kept in its descriptor. The bit needs to be zeroed when an LWT is running, when it is blocked and when it exits. We choose to make the first transition eagerly: When the worker picks an LWT to run, the corresponding bit is in the worker's cache, so that marking the bit then is efficient. The bit is not zeroed immediately when an LWT blocks or exits. It is changed to one as soon as a thread yields. This policy optimizes case where LWTs execute to completion or are suspended only if blocked, waiting for an event. The use of yield while an LWT is not required to suspend is inefficient and is often used to enable polling by ohter LWTs for the completion of events. The support of event-driven scheduling reduces the need for polling. It is worth noticing that most OpenMP implementations implement OpenMP `taskyield` as a noop.

Algorithm 4.1 describes this scheduling strategy. We assume a bit-vector is implemented as an array of 64-bit values. Each time a worker traverses this array, it makes a copy of a specific word and also exchange it with a 0 value, clearing the bits in that word. The scheduler will then execute in sequence all the LWTs with a bit set in this word. This is done by repeatedly finding the first non-zero bit and flipping it, until the entire word is zero. The actual LWT is found in the array of LWTs using the bit and word indices. The use of atomic exchange for a word is a practical decision, since most of the modern machine atomic instructions work better at the level of 64-bit word; and further, the copy of the word is now local to the worker thus avoiding cache misses and preventing data races that might occur when another worker tries to modify the bit-vector (for remote signaling or work stealing).

Algorithm 4.2, 4.3 describe the implementation of `Spawn` and `Yield`. When a new LWT is spawned, a free bit is obtained from a pool of free indices. At this point, to make sure the LWT will be picked up by the worker, we set the associated bit in the bit-vector. The `Yield` is simpler: We set the bit before switching to the scheduler's code. For simplicity, we assume a `Join` operation is implemented by `Yield`-ing until the LWT being joined completed; we shall later optimize the `Join` operation.

*There are no data races in the scheduler if **`GetBit`** and **`SetBit`** are thread-safe.* The only possible interaction between workers, without work-stealing and remote signaling, is when an LWT is spawned at a remote worker. The spawn requires two calls, `GetThread` and `GetBit`. The former is implemented by a concurrent pool, while the latter is implemented via an atomic bit set – which is provided by most of the modern machines. When an LWT exits, its bit is garbage collected to the pool of free indices.

No strong fairness condition such as FIFO is guaranteed (later in Section 4.2.3 we show how it can be approximated if needed with an appropriate implementation). *The baseline algorithm maintains a progress property: a spawned or runnable LWT will be eventually executed or resumed.* This property is enforced via the way we traverse the bit-vector. To show that this is true, the atomic exchange can be considered as taking a snapshot of the global state. In this snapshot, if an LWT is marked, it will be scheduled after all LWTs having smaller indices are scheduled, which is bounded by the total length of the bit-vector.

4.2.2 Signal/Wait Synchronization Primitive

We add now the following two synchronization primitives to the baseline algorithm:

- `Wait()`: Suspends the current LWT without marking it runnable i.e. leaving the bit as 0; the LWT is blocked.
- `Signal(t)`: Sets the bit associated with the LWT t as 1, making it runnable.

`Signal(t)` marks LWT t as runnable. The signal can occur either before or after the target LWT calls `Wait` — this avoids the need for mutual exclusion in the producer-consumer synchronization protocol – with a significant reduction in cost. On the downside, the signal may be lost if the target LWT yields (with a call to `yield`, rather than `wait`), after the signal was raised. Thus, we decree that the behavior of `Signal(t)` is undefined if the signal call is concurrent with a `Yield()` call executed by t. This is a minor inconvenience, in practice. Behavior is also undefined if the signaled LWT has terminated or if no `Wait()` call ever matches the `Signal(t)` call. Multiple concurrent `Signal(t)` calls to the same LWT have the same effect as one call.

The implementation is straightforward if `Signal(t)` is only invoked by the worker that owns LWT t. If `Signal` can be called by another worker, a data race can happen; therefore, bit updates need to be done using atomic operations. The atomic bit update can be done with either `fetch_and_or` or atomic bitset instruction which are provided by most modern machine.

There is no need for updating the state of the LWT when its bit is modified, even though a 0 is associated with multiple states. The scheduler only cares whether an LWT is runnable or not, and does not care whether the LWT cannot be scheduled because it is invalid, is already running, is blocked, or has terminated.

By retaining at most one premature signal, one can solve the producer-consumer problem without mutual exclusion. The pseudo-code for such solution are shown in Listing 4.4. Since `SetFlag` happens before `Signal`, then the only possible data race is when the `Signal` happens in between Line 7 and Line 8. Since that signal is retained in our scheduler, the thread will eventually be scheduled. The while loop in the code is necessary if the thread can receive signals other than from the consumer, creating spurious wakeups [108].

Listing 4.4: FULT Wait/Signal solution to the RSPC; ”s.thread” is the consumer thread

```
1 Producer(s)
2   PlaceItem(s.buffer)
3   SetFlag(s.flag)
4   Signal(s.thread)
5
6 Consumer(s)
7   while (!s.flag) do
8     Wait()
```

4.2.3 Supporting Generic Tasks and Synchronizations

We assumed in the previous section that the bit-vector is short so that it can be traversed with low overhead. The length of the vector is limiting the number of concurrent LWTs, but not the total number of LWTs spawned during execution. For many applications a limit of 256 LWTs per worker is adequate, but some systems may require larger LWT counts; the inability to spawn a new LWT until some other LWT completed may cause deadlocks.

We can support a larger number of concurrent LWTs at modest cost. The idea is to use a hierarchical bit-vector structure. More specifically, the first-level bit-vector can be used as a “hint” to index into the next level bit-vector. That is, each bit in the first level bit-vector indicates which bit-vector at the next level may have a set bit. A `Signal` sets the associated bit in each level from bottom to top, while a worker schedules by traversing the structure from top to bottom.

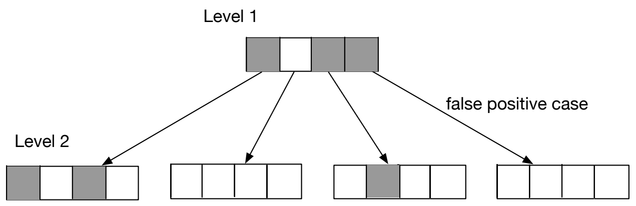


Figure 4.2: An illustration of 2-level bit-vector, where each bit of the upper level approximates a group of 4-bit in the lower level.

It is easy to see that *there is never a persistent state where the upper level and lower level are inconsistent.* Such transient situation can occur when a bit is flipped at one level and is not yet flipped at the other level. Since the top level is considered to be only a hint, a transaction is committed when the lower level bit-vector is modified, and these changes are atomic. To prevent races and improve performance, we flip bits from 1 to 0 lower level first, higher level next; and flip bits from 0 to 1 higher level first and lower level next. Figure 4.2 illustrates this data structure, and the possible "false positive" case.

Suppose we can tolerate up to two cache misses to find a runnable LWT, and each cache line is still 64-byte. The first level can contain 512 bits, each bit hints to a bit-vector with length 512-bit in the second layer. Then in total we can afford up to $256K$ concurrent execution LWTs per worker. A system with 64, 4-way multithreaded cores will support more than 16M concurrent LWTs, more than enough for practical applications.

Load-balancing with work-stealing A common algorithm for load balancing across a large number of workers is *work-stealing* [109, 110]: Each worker maintains its own work list; spawned tasks are added to the local list and tasks are picked for execution from the local list. However, if a worker is left with an empty list, it then "steals" a task for execution from the list of another randomly chosen worker. While it is sufficient to steal one task at a time, better performance is obtained if more tasks are stolen at a time.

Hendler and Shavit [111], following on previous work by Rudolph et al. [112] analyze a probabilistic *steal-half* procedure whereas workers probabilistically decide whether to steal based on the workload; if they decide to steal, then they pick a random worker and, if the chosen worker has more tasks, they pick half of this surplus. This strategy ensures that, at any point in time during the execution, the expected number of tasks at any worker is at most a constant factor larger than average number per worker, provided that only a fixed number of new tasks can be generated at a worker in between steal attempts. This also imply that the expected total computation time will be $\mathcal{O}(W/p + S)$, where W (work) is the total number of operations performed, and S (Span) is the critical path length of the

Algorithm 4.4 Worker scheduler with Work-Stealing

```
procedure SCHEDULER($\omega, V$: corresponding worker and its bit-vector)
    while !$\omega$.stop do
        hasWork := 0
        for word in $V$ do
            if word $\neq 0$ then
                localWord := 0
                AtomicExchg(word, localWord)
                while localWord $\neq 0$ do
                    b := Leadingbit(localWord)
                    localWord := ClearBit(localWord, b)
                    t := GetThread($\omega$, b)
                    ContextSwap($\omega$, t)
                    hasWork := 1
                end while
            end if
        end for
        if !hasWork then
            $\omega'$ := RandomWorker()
            Steal($\omega$, $\omega'$)
        end if
    end while
end procedure
```

computation.

The usual data structure used for work list is that of a dequeue: the local scheduler manages the list in FIFO order, while stealing workers access the other end of the list. However, the previous theorem holds for any structure that ensures that stealing has a constant overhead. In our case, a thief directly works on the bit-vector of the victim until it has found a word with runnable LWTs, which in most cases is a constant time operation. While we do not follow the exact probabilistic strategy that was analyzed in the referenced papers, work-stealing algorithms tend to be robust and experiments show that our variant has the desired behavior.

The critical issue is to avoid data races and contention when multiple workers attempt to steal from the same list. Data races are avoided as work-stealing uses the same atomic operation as the local scheduler, namely atomically copying a word from the bit-vector into a private word, and zeroing the entries in the bit vector. Congestion could occur if multiple workers try to grab the same word. Congestion does not seem to be a problem, in practice, as a thief will continue to the next word on an unsuccessful swap instead of retrying on the same one. Algorithm 4.4 and 4.5 shows the scheduler code work and the steal routine for a

Algorithm 4.5 `Steal`(ω, ω')

```
1: procedure STEALING($\omega$: thief or current executing worker, $\omega'$, $V'$: corresponding victim
   and its bit-vector)
2:     for word in $V'$ do
3:         if word $\neq 0$ then
4:             localWord := word
5:             AtomicExchg(word, localWord)
6:             while localWord $\neq 0$ do
7:                 b := Leadingbit(localWord)
8:                 localWord := ClearBit(localWord, b)
9:                 t := GetThread($\omega'$, b)
10:                ContextSwap($\omega$, t)
11:                return
12:            end while
13:        end if
14:    end for
15: end procedure
```

1-level thread scheduler in the absence of `Signal` calls.

Work stealing requires additional work if signals are supported: A 1 bit may indicate an LWT that is running but was signaled. Such an LWT should not be stolen. We handle this situation by having the thief reset one bits in the original bit-vector for all LWT's for which this situation occurs. This can be done in one operation that resets all needed bits in the target word, using a suitable mask and an atomic `fetch&or`. However, in a contended case, it is still possible that two workers find the same 1-bit and try to run the same LWT. Therefore, whenever work-stealing is needed, a worker performs `compare&swap` (`CAS`) on an additional status flag of the LWT object to break the tie and ensuring that only one worker will execute the LWT.

Another more elegant solution for the above problem is to use more than one bit to fully represent the state of an LWT. It is possible to encode the states so that both signaling and stealing can be executed with one atomic `fetch&and`. We plan to explore this method compared to our current implementation for performance in future work.

In our design, work-stealing does not imply LWT migration. The ownership of the LWT returns to the original owner when the LWT yields or blocks. (However, if it spawns children, the children are owned by the thief.) With such an approach, an LWT is permanently associated with a bit in a bit-vector, and a signal can directly target this bit.

We can also support migration, in which case, a signal will require an indirection, for checking the bit location in the LWT descriptor. As we allow signals to be lost if the target thread yields, it is possible to permanently migrate a stolen LWT at the point where it executes a yield, without worrying about signal redirection. This simplifies the migration procedure.

A migration then consists of the following steps:

1. Allocate a new bit in the bit-vector of the thief.

2. Update the LWT descriptor to point to the new bit.

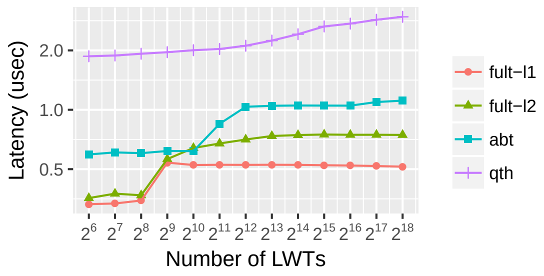
3. Mark the new LWT as runnable.

A signal may go into the old location while the migration is happening during step (2), this signal is therefore lost, which is consistent with our definition of *signal* before a *yield*. After step (2) is done, the migrated LWT should now have the new information which points to new worker, thus any new signal will go to the correct location.

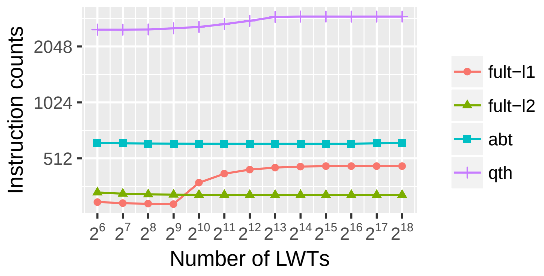
Spawn and Join Optimization Since sometimes FIFO/LIFO order matter for certain problems, the scheduler can be configured to approximate this behavior. The trick is to control the order in which bits are allocated and the order of traversing the bit vector. First, the index of the bit assigned at each `Spawn` keeps increasing until we are out of bit indices or when some bit indices are recycled. Second, if depth-first is preferred, a worker traverses its bit-vectors from the largest bit back to smallest; and vice-versa for breadth-first. The default scheduler, which we used in the experiment, approximates a depth-first scheduler (similar as Cilk), which has a good provable bound on memory space [106].

The implementation of `Join` can be optimized when an LWT attempts to join its own children. This is a direct application for `Wait/Signal`. In this approach, each LWT keeps a pointer to its parent so that when the LWT is exiting, it can perform a `Signal`. Further, when the parent and the child are from the same worker, instead of `Signal`, the worker can start working directly on the parent. When an LWT spawns and joins more than one child, an additional counter can be used to avoid signaling the parent multiple time. Note that this optimization fits well with to our scheduler since our wait/signal is very efficient. Qthreads uses its Full-Empty Bit synchronization object for joining which is directly comparable to our optimization.

Other Synchronizations Although not encouraged, LWTs might have to support mutual exclusion. We provided two types of locks. One is `spinlock` which is implemented using a simple `test-and-set`. The user should use this lock only if conflicts are rare. It is erroneous for an LWT to hold a `spinlock` while suspending. The second type of lock is `qlock` which



(a) Performance in terms of latency



(b) Performance in terms of Instruction count .

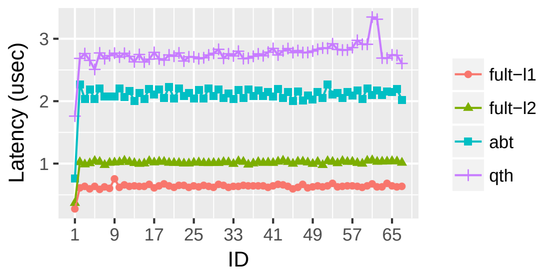
Figure 4.3: Performance of FULT Spawn and Join with one and two-level bit-vector (fult-1l, fult-l2), in comparison to Argobots (abt) and Qthreads (qth)

is more integrated with the scheduler and makes use of `spinlock` for the implementation. This lock allows the LWT to perform `Wait` while being recorded in a FIFO queue. The owner of the lock shall `Signal` one thread in this queue once it has finished. The `spinlock` is required to avoid data races when some LWTs are being added the queue, while the owner LWT is releasing the lock. A counting barrier is also implemented atop of `qlock` and a counter. Since a queue is involved in both of these objects, the implementation of FULT is very similar to that of Qthreads lock and barrier based on Full/Empty Bit. We later show that we get better performance due to the more efficient Signal/Wait.

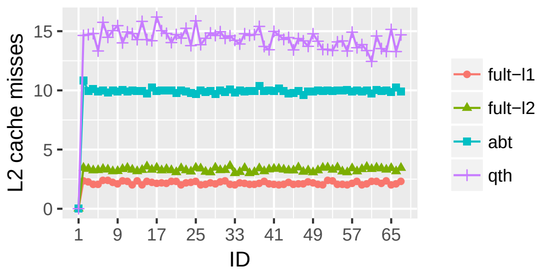
4.3 PERFORMANCE EVALUATION

All of our experiments in this section were done on the Texas Advanced Computing Center's Stampede2 KNL Cluster (Stampede2). A KNL compute node consists of 68 cores with a speed of 1.4Ghz; the fast memory is configured in cache mode, providing 16GB L3 MCDRAM cache. Code is compiled using Intel `icc` compiler version 17.0.4. We mainly compare FULT with Argobots git commit `8e1588` and Qthreads git commit `930314`; both are the latest at the time of the experiment. Unless specified, a worker in each of the thread systems is bound to a core, and each worker works on its own private run-queue (i.e. *shepherd* of Qthreads, *pool* of Argobots). The scheduler of Qthreads uses work-stealing whereas Argobots performs load balancing by assigning LWTs to workers in a round-robin manner. The profiling information (such as cache misses) is gathered using the PAPI toolkit available on the system [113].

We evaluate the runtime overhead for scheduling, by spawning large numbers of LWTs on a single worker; each LWT does no work and simply returns immediately. The procedure is done 1000 times, and we show the average overhead per LWT spawning for all the experi-



(a) Performance in terms of latency



(b) Performance in terms of L2 cache misses

Figure 4.4: Performance of FULT signal/wait with one and two-level of bit-vector (fult-l1, fult-l2), in comparison to Argobots (abt) and Qthreads (qth).

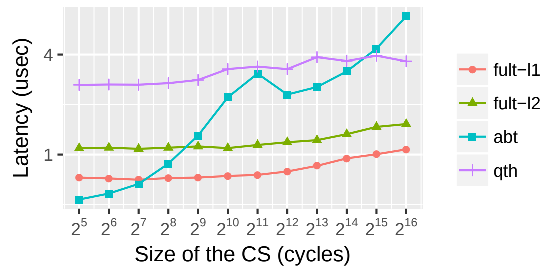
ments, Figure 4.3(a) shows the results of this experiment. FULT using one-level bit-vector is the best in all tested cases, follows by the two-level, Argobots then Qthreads.

Argobots used an Yield-based approach with many optimizations, while Qthreads uses a Full-Empty Bit for Joining. Both requires more instructions to spawn and join a thread than in FULT. This is verified by the instruction count shown in Figure 4.3(b). FULT using one-level bit-vector is upto 2× better than Argobots and 6× better than Qthreads. The decrease in the performance of FULT with a one-level bit vector occurs at 512 threads. This is when the one-level bit-vector is filled up and FULT has to continuously recycle the LWT bits. At that point the two-level structure becomes more efficient.

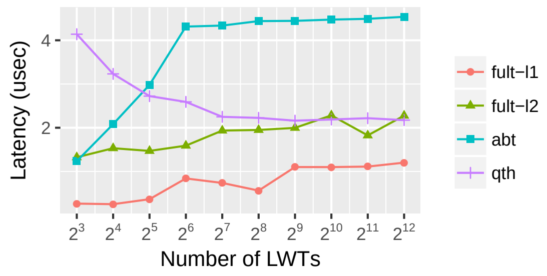
4.3.1 Wait and Signal

This experiment evaluates the effectiveness of our Signal/Wait mechanism, compared to the synchronization primitive in Argobots and Qthreads. The benchmark is a shared-memory ping-pong between two LWTs spawned in two workers, one bound to core 0, and the other bound to one of the other cores in the KNL. This is similar to the producer-consumer code in Listing 4.1: One LWT sets a flag and notifies the other LWT then waits for the flag to be reset. Vice versa, the other LWT will reset the flag, notifies back and then wait for the flag to be set. The procedure is executed 1000 time and the average is shown.

The average latency for each Signal/Wait are shown in Figure 4.4. As expected, FULT outperforms both Argobots and Qthreads in all cases. To verify that our scheduler reduces cache misses, we also measure the number of misses in the shared L2-cache and the results are shown in Figure 4.4(b). Signal/Wait in FULT causes at most 2-3 cache misses in both cases, where as it can cost 10 to 15 cache misses in Argobots and Qthreads, which explains the difference in performances.



(a) Performance of FULT qlock



(b) Performance of FULT barrier

Figure 4.5: Performance of FULT qlock and barrier with one and two-level of bit-vector (fult-l1, fult-l2), in comparison to Argobots (`abt`) and Qthreads (`qth`)

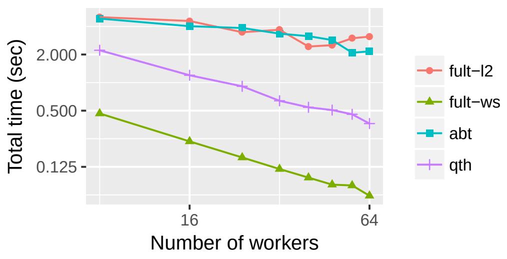
4.3.2 Lock and Barrier

We compare in Figure 4.5(a) FULT's `qlock` with Argobots mutex (`ABT_mutex`) and Qthreads' lock (`qthread_lock`). The experiment spawns 64 LWTs and assign one to each of 64 workers. Each LWT loops 1000 iterations where each iteration is a sequence of: lock, critical section (CS) and unlock. The CS code is a computation which requires the number of cycles shown on the x-axis. The total time is measured when all the LWTs have returned. The latency of Lock/Unlock is plotted in y-axis, computed by subtracting from the total time the time consumed by the CS and averaged by the number of iterations.

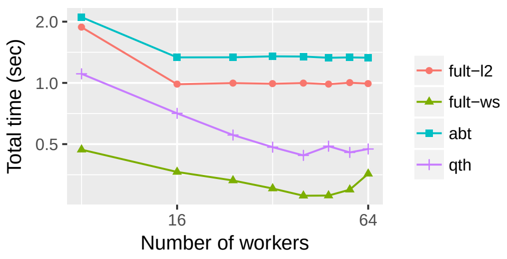
When the CS is short, Argobots performs best, since its implementation uses a `Yield`, which does not require the extra latency of adding a thread to a queue. FULT `qlock` performs better than Argobots with a CS of more than 128 and 256 cycles using the one-level (fult-l1) and two-level (fult-l2) scheduler, respectively. Qthreads also uses Signal/Wait via the Full-Empty Bit for their lock, thus being slower than both. When the CS becomes larger, such as at $32K$ cycles, Argobots becomes the worst, followed by Qthreads, while FULT remains the best. At 1024 cycles, FULT fult-l2 is 2x faster than Argobots and 3x faster than Qthreads. At $32K$ cycles, FULT fult-l2 is 3x faster than Argobots and 2.7x faster than Qthreads.

Figure 4.5(b) shows the evaluation of a barrier in FULT in comparison to Argobots (`ABT_-barrier`) and Qthreads (`qt_barrier`). In this experiment, we spawn a number of LWTs, assigned round-robin to each of the 64 workers. An LWT loops 1000 iterations, each time it waits on a pre-initialized barrier object that accepts the same number of spawned LWTs. The total time is measured when all LWT has returned, and averaged by the number called barriers to get the plotted latency.

When the number of LWTs is smaller than the number of workers, Argobots and FULT perform better than Qthreads. When there are more LWTs, only fult-l1 outperforms Qthreads.



(a) Computing Fibonacci number 32



(b) Solving nQueen problem with problem size 11

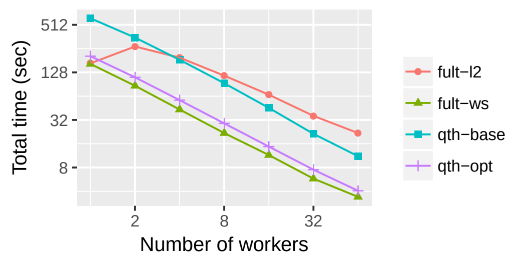
Figure 4.6: Work-stealing evaluation of FULT two-level bit-vector (fult-l2), and that with Work-stealing (fult-ws) in comparison to Qthreads (qth) and Argobots (abt)

We believe this is when the locality of the queue pays an important role, and becomes the bottleneck for both Qthreads and FULT fult-l2. Similar to its lock implementation, Argobots uses an algorithm based on Yield, and thus appears to be not as efficient.

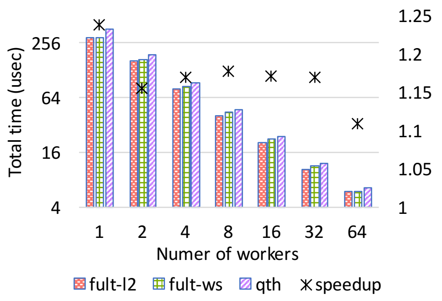
4.3.3 Work-Stealing

The Work-stealing implementation in FULT is compared to that of Qthreads. Since Argobots does not support work stealing, we use its scheduler together with the 2-level with no stealing to show the performance without work-stealing. In this case, we simply assign LWTs to workers round-robin. We use two typical problems that were used to evaluate work-stealing: the computation of a Fibonacci number without memoizations and the computation of the number of ways to places n queens on an $n \times n$ board. The two benchmarks were previously implemented with Argobots and Qthreads (the reference implementations can be found in the Qthreads package and the GLT benchmark [114]). In both benchmarks, a large number of LWTs are created when the algorithm is running.

The performance comparisons are shown in Figure 4.6(a) for the Fibonacci problem and Figure 4.6(b) for the n-queen problem. In both cases, we can see that without work-stealing the performance is not scalable. When work-stealing is used, the FULT scheduler is superior to Qthreads, up to 6× faster. In both cases, aside from the ability to balance the workload, the performance relies on fast Spawn and Join, where FULT was shown to be better. FULT's work-stealing however shows degradations when there are not enough works to split amongs all the workers such as in the case of n-queen with 64 workers. Thus, we believe there are still rooms for more advanced strategies.



(a) Unbalanced Tree Search (UTS) strong-scaling using the T3L tree.



(b) HPCCG strong-scaling runtime using input 256^3 (left axis) and speedup (right axis).

Figure 4.7: Performance of mini-applicatiosn with FULT two-level bit-vector (fult-l2) and that with Work-stealing (fult-ws) and Qthreads (qth)

4.3.4 Mini-Applications

UTS evaluates the performance of parallel systems under heavily unbalanced and irregular workloads [115]. The benchmark generates a randomized tree based on sampling from a configured probability distribution and an initial seed until no more nodes in the tree are generated. Our implementation is based off the simplest Qthreads implementation in their source package (`uts_aligned`). We compare it to a more optimized version of Qthreads implementation which used a counter to improve their Signal/Wait mechanism (`uts_qthreads`). The average of 5 runs for the input tree $T3L$ of the package (10^8-nodes binomial tree) is plotted in Figure 4.7(a).

HPCCG is a miniapp which represents an approximation to a finite-volume problem. The code for Qthreads are based off the Mantevo benchmark suite [48] and is included in its package. Qthreads version implements loops in the original code using *task loop* (iterations are divided in chunks and processed in independent LWT). FULT adopts an algorithm from the Qthreads implementation of `qt_loop_balance` to spawn task loops and uses that to run the application.

In both cases, FULT achieves a better performance than the optimized Qthreads implementation, though with a simpler code. This is because the default Spawn/Join implementation in FULT is more performing, thus no further optimization is required. The behavior of the non work-stealing version fult-l2 shows that FULT work-stealing performs well under heavy workload, similar to that provided by Qthreads.

4.4 RELATED WORKS

Low-overhead context-switching is an important feature for High-Performance Computing systems, especially when a large number of concurrent threads are desirable. Many threading models and libraries have been developed to leverage this benefit of LWT such as [116, 117, 118, 119, 120, 121, 122, 114].

Though using a similar mechanism for context switching, each of the threading libraries is optimized for different purposes. Converse threads [118] are designed for the Converse framework, incorporating a hierarchical scheduling model. MassiveThread [120] is optimized for recursive task parallelism. GLT [114, 123] studied and implemented an efficient, portable and sufficient unified API for implementations of LWT; FULT operations are motivated by this API. Qthreads [122] is optimized for Full/Empty Bit synchronization with advanced work-stealing and was used to implement task-based system with distributed memory such as Chapel [124].

Argobots [121] is one of the most recently developed LWT system and also, the most flexible: with reasonable understanding of the API the user can customize almost every detail of the threading system without touching Argobots' source code. Argobots was built for the Argo operating system project [125], and is also used for communication systems [126, 58]. Argobots original paper and a later one in [127] show that the baseline was optimized and outperformed other threading system; the only downside is that it does not have a built-in work-stealing mechanism.

FULT was inspired by Qthreads and Argobots and has taken advantage of many well-known optimizations studied by other threading systems. To our best knowledge, FULT is the first threading system that represents runnable LWTs using bit-vectors and utilizes this structure to reduce the cost of signals to a minimum and simplify their logic to the point that a hardware implementation is very feasible.

4.5 SUMMARY AND IMPACT

In this chapter, we analyze FULT, an LWT scheduling technique using bit-vectors. The performance advantages come from the very efficient Signal/Wait which avoids both mutual exclusion and the moving of threads' metadata between concurrent queues for many cases. When using bit-vectors, scheduling and load-balancing can be challenging at scale; we have presented effective solutions to these problems. Overall results have shown that the FULT scheduler is often better than other LWT schedulers that it is based off, such as Argobots and Qthreads. The difference is most significant when work stealing is not required and when signal-wait synchronization is frequently used.

The FULT scheduler open opportunities to includes an integration into OpenMP task scheduler to provide efficient inter-operation with communication layer such as MPI. We also expect to further reduce the overhead of work stealing by using lower-cost mechanisms and improved policies. The work in this chapter was published at [128].

In the next chapter, we finally describe the design and implementation of LCI, and demonstrate that with the help of FULT, concurrent message-passing can be truly efficient.

CHAPTER 5: EFFICIENT MODEL FOR CONCURRENT MESSAGE-PASSING

Many challenges has arrived in both programmability and performances of MPI having to cope with changes in the new architectures [129, 130]. For this reason, many projects have explored the use of a partitioned global address space (PGAS) across nodes [131, 132], the use of fork-join models with lightweight threads [133], and coarse-grain dataflow models within and across nodes [102, 104].

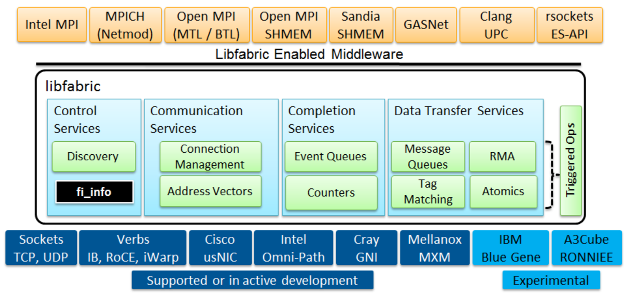


Figure 5.1: Libfaric communication library (figure taken from [91]).

Emerging system softwares implement their communication subsystem using either GASNET, MPI or OpenSHMEM [134]. Even though both communication libraries support a rich set of semantics, they are heavyweight and does not inter-operate very well with customized library. Further, they have to be refactorred, to more easily adapt with newly design intelligent NIC, and the new multi-core architecture.

UCX [30] and Libfaric[91] are new developments from the industry to assist traditional softwares like MPI. Both provide rich functionalities and have great flexibility (Figure 5.1 shows the overview of Libfaric). Even though they can potentially be used directly by application frameworks, the major users are limited to those that mentioned.

Since the idea is to support traditional users first, kernel threads and kernel signaling mechanism such as file-descriptor and Pthread condition variable are typically used (UCX *async event*, Libfaric *Waitset*). At the time of our study, it requires significant software stack to extend the support for other light-weight tasks or customized libraries.

We believe in a need for a *re-targettable communication interface*, that can be specialized towards new hardwares and also optimizable for specific distributed algorithms (as has been shown for Graph analytics). This chapter studies the requirement for many frameworks, and describes the design as well as implementation of our model.

5.1 OVERVIEW OF RECENT COMMUNICATION FRAMEWORKS

The Open Community Runtime [135] project aims at building a runtime to support *Asynchronous Many Task* (AMT) systems for large scale computers. The work on OCR focused on higher-level functionality (maintaining tasklets and their dependences, scheduling them for execution and managing allocation of memory and data migration) with an emphasis on shared-memory parallelism. Proof-of-concept for distributed memory versions using existing communication libraries (MPI and GASNET) has been made.

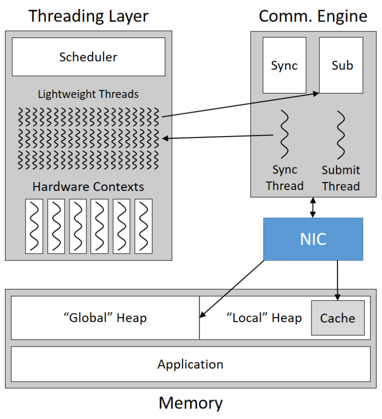
HPX is an effort to build a runtime that incorporates a well defined execution model, ParalleX [136]. This model is quite different from what is now natively provided by current hardware. It assumes an "Active Global Address Space", a parcel subsystem for the transfer of data and control, a threading subsystem, and Local Control Objects (LCOs) for synchronization. HPX provides a very good testing ground for the ParalleX execution model. It is less clear how well it matches the capabilities of the forthcoming extreme scale systems or the needs of emerging event-driven programming models.

Legion [104] provides notation to organize data into logical regions that can be partitioned into subregions. Tasks specify which regions they need to access and what access privileges they need (read, read-write, or reduce). The location of data (including possible replication) and the scheduling of tasks is under runtime support. The runtime attempts to collocate data and task execution so as to minimize communication, and schedule tasks in an order to respect their dependencies.

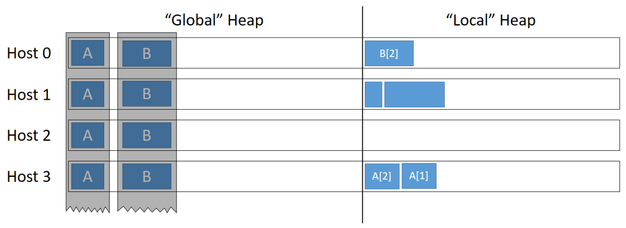
Parsec [105] uses a dataflow execution model: a program specifies tasks (functions), a dataflow graph connecting the tasks, and hints about data distribution. The runtime statically allocates tasks and data to nodes. The actual execution of tasks is event-driven, with a task being scheduled on an available core when its dependencies are satisfied. The model handles distributed heterogeneous platforms (tasks executed on GPUs).

PPL [137] is a modular, distributed-memory, C++11 runtime library for exploring the needs of future programming models on large-scale systems and rapidly testing different configurations. PPL consists of three primary components as seen in Figure 5.2(a): *a communication engine*, *a memory layer*, and *a threading layer*. It uses a PGAS memory model atop a custom communication library that directly uses network-level one-sided RDMA operations. Multi-threading is natively integrated into PPL and is targeted toward lightweight task, massive concurrency, and fine-grained synchronizations. PPL incorporates a transparent software-based cache for remote memory accesses, which enables synergistic caching, where many threads share the same remote data.

A common feature of these above frameworks is a polling agent for communication engine



(a) The PPL architecture.



(b) The PPL memory layout.

Figure 5.2: The PPL design and memory layout. Objects can be created independently on the local heap or collectively on the global heap (e.g. A, B are created on the global heap). The local heap is also used to cache remote data (e.g. $B[2]$ caches object B from host 2 in memory of host 0).

which has two functionalities: handling communication requests and performing associated synchronizations. Other threads (maintained by a threading layer) perform communication through the engine by submitting requests into a software queue.

In PPL, the communication uses two threads. The *Submit* thread dequeues requests from this queue and submits the corresponding communication to the network. When there are no pending communication requests, the *Submit* thread polls the network for completed communication and pass it to the *Sync* thread via a private single-producer, single-consumer queue for triggering associated synchronization. This design can be improved by reducing a thread, with faster thread synchronization.

Parsec uses a dedicated communication engine for driving the communication implemented with MPI. It emulates one-sided communication using two-sided communication and persistent requests. It also applies flow-control through the use of request queues and issuing communications from a single thread. These overheads can be reduced by directly using RDMA with a more relaxed version of MPI one-sided communication semantics.

Legion implemented its communication runtime using Active Messages model with GASNET. Since Legion uses a light-weight task-based scheduler, GASNET active handler has significant overheads for enabling tasks and moving data between the communication buffer and the tasking layer. The overheads come from the execution of handlers, which has to pass control to another execution entity due to its restrictions. Further, the handler cannot be inlined, the data buffer is usually owned by GASNET which causes additional copies.

Both OCR and HPX does not expose explicit communication, but they can be benefit

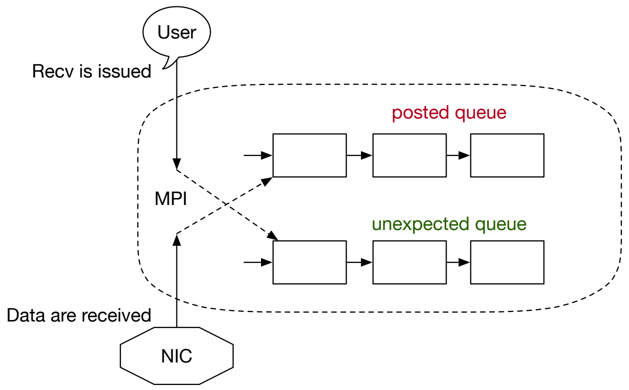


Figure 5.3: MPI runtime message-matching with two queues. When receive request is issued by the user, MPI has to find a matching request in unexpected queue; when the data is arrived from the NIC, MPI has to find a matching request in the posted queue. If the match cannot be found in both cases, MPI appends to the other queues.

from going through a lower-level communication interface, which are closer to hardware.

Other common features that should be supported can be summarized:

1. Allows concurrent LWTs to execute communication code more efficiently, to avoid the need for funneling requests issuing.

2. Support for lightweight threads (tasks) with fast synchronization primitive: fast task activation and preemption to avoid busy-waiting but still hide latencies.

3. Support for low-level network capabilities such as RDMA while bypassing other complex communication semantics, which reduces the software overheads of the runtime.

4. Support for event-driven task scheduling in a generic way by allowing multiple producer-consumer patterns and communication signaling methods.

In the subsequent sections, we describe a system which provides: (1) by revisiting and relaxing the message-matching semantics of MPI; (2) by tightly coupling the thread primitives with the communication interface; (3) by using RDMA directly in the communication protocol and (4) by a generic APIs that is able to support multiple patterns of communication.

5.2 SYSTEM FOR CONCURRENT MESSAGE-PASSING

In MPI, the mechanism for matching sends to receives in the right order is to use a linked list for posted receives and a linked list for early arriving (unexpected) sends. When a message arrives, it is paired with the first matching receive in the posted receive list; if none is found, it is appended to unexpected send lis. Symmetrically, when a receive is posted, it is paired with the first matching send in the unexpected send list; if none is found, it is appended to the posted receive list. Figure 5.3 illustrates this design.

With an increasing number of threads, there can be a larger number of concurrent sends or receives, hence longer lists are searched sequentially and concurrently updated; the sequential search and update operation must be atomic — leading to higher contention or more complex protocols. Various attempts have been made to minimize this bottleneck by partitioning the range of communicator, sender and tag values (and using distinct lists for each partition). This is easy for communicators, but traffic is usually not balanced across communicators.

The use of wildcards complicates the approach for sender and tag values further. One either must insert wildcard receives in multiple queues, or maintain a separate queue for wildcard receives that needs to be searched concurrently with the regular one (see discussion in [138], where a partial solution is provided). An alternative approach is to relax the restrictions of MPI semantics. For example, the MPI forum is currently discussing a proposal that would enable programmers to disable wildcards or relax ordering semantics, for selected communicators [139].

In order to support a highly concurrent message-passing protocol, LCI uses the extreme approach by prohibiting the use of wildcards. We show that when that is the case, the performance difference between single-threaded and multi-threaded communication largely disappears.

5.2.1 High-Level System Design

We propose an implementation of message-passing in LCI based on the following assumptions:

- Large number of concurrently communicating threads. Threads are lightweight; they are scheduled by a user-space scheduler that handles blocking synchronization. The number of light-weight threads may be significantly higher than the number of physical threads, and over-decomposition may be used to hide communication latency.
- Large number of cores; it is possible to dedicate one or more physical threads to communication.
- The Network Interface Controller (NIC) can be accessed in user space; it has its own routing tables, in order to translate ranks in MPI_COMM_WORLD to physical node

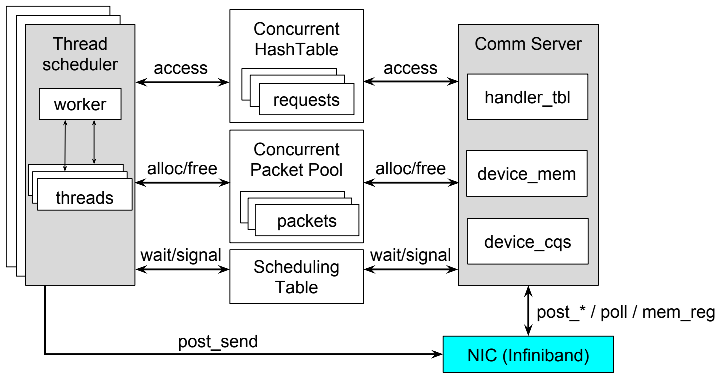


Figure 5.4: Runtime architecture for multi-threaded message-matching

addresses; it has page tables, in order to translate virtual addresses to physical addresses. We consider in our work InfiniBand and OmniPath adapters, but the design should port to other adapters.

- We consider only x86_64 architecture; however, the technique is general enough to port to other architecture that supports atomic exchanges and atomic bit manipulations.

Figure 5.4 shows the overall architecture of our described runtime system.

We use a dedicated kernel thread as a *communication server*. This communication server executes all the communication protocol that is asynchronous w.r.t. the communicating *workers*, such as polling and handling the rendezvous protocol. Ideally, the communication server logic could be executed directly by the NIC. This design reduces contention and reduces cache pollution at computing threads.

The workers execute *Light-weight Tasks* (LWT). These are managed by a LWT scheduler. The scheduler is simply a function invoked when a LWT completes or yields. A LWT executing a blocking MPI call will yield, and invoke the LWT scheduler; the scheduler will schedule another LWT on that worker. We assume in this chapter that the association of ULTs to workers is fixed: ULTs are not migrated once they started executing.

The communication server and the workers share three data structures. Our simple design allows optimizations to be focused on these three critical shared data structures and operations on those:

- A *hash table* that is used to match sends and receives.
- A *scheduling table* that is used to mark which LWT are runnable.
- A *packet pool* for packets posted to the NIC.

The hash table stores both unexpected incoming messages and outstanding receives. Since we assume there are no wildcards, we can hash by key (`<communicator, sender, tag>`). Furthermore, our assumptions imply that each communication will involve one insert in the hash-table, when no matching is found, and one delete, when a match is found. The insert is for the first occurring operation (either a receive or an unexpected send); the delete is for the second occurring operation of the send-receive pair.

The scheduling table is used by the communication server to mark a LWT as runnable when a communication operations it is waiting for has completed (it is also used for thread synchronization).

The packet pool supports two basic operations, namely `alloc` and `free`, which obtain and return a packet from/to the pool respectively. The management of request records (more relevant for non-blocking calls, blocking calls can use stack space), is similar but simpler than the packet pool as it does not require flow control. Thus, we omit its discussion for the rest of the chapter.

5.2.2 Algorithms and Protocol Details

Algorithm 5.1 Eager-message send/recv for thread

```
 1: procedure SEND-EAGER($b, s, k$)                    ▷ : buffer, size, key=<dest, tag>
 2:     $p$ = pkpool.alloc()
 3:     Set packet header $p.h$ to $k$
 4:     Copy $b$ to $p.b$
 5:     Post $p$ to network.
 6: end procedure
 7:
 8: procedure RECV-EAGER($b, s, k$)                    ▷ : buffer, size, key =<from, tag>
 9:     Create a request $v = (b, s, t)$               ▷ buffer, size, thread id
10:     $v' = H$.access($k, v$)
11:     if $v' \neq \perp$ then                        ▷ :match found
12:         Copy $v'.p.b$ to $b$                       ▷ : message arrived, copy data.
13:         pkpool.free($v'.p$)
14:     else                                           ▷ : insertion success
15:         ThreadWait()                               ▷ : message not arrived, wait.
16:     end if
17: end procedure
```

Message delivery is implemented in two ways: *eager* or *rendezvous* protocol. Eager protocol is used for short messages: The message header and content are copied into a packet

Algorithm 5.2 Eager-message packet handler for communication server

```
 1: procedure RECV-EAGER-PACKET(p)
 2:     $v' = H.\text{access}(p.k, p.v)$
 3:     if $v' \neq \perp$ then                                ▷ :match found.
 4:         Copy $p.b$ to $v'.b$                  ▷ : message arrived, copy data.
 5:         ThreadSignal($v'.t$)                   ▷ : mark receiver as runnable
 6:         pkpool.free($p$)
 7:     else                                              ▷ : insertion success.
 8:         return                        ▷ : message not arrived, nothing to do.
 9:     end if
10: end procedure
```

that is delivered to the network. The Send operation returns immediately as the send buffer can be reused. This protocol becomes inefficient when message size gets large. When this is the case, we switch to the rendezvous protocol in which the data is delivered directly from the source buffer to the target buffer by the NIC, thus saving extra copies. The rendezvous protocol requires additional messages to exchange control data and signal completion.

An eager protocol is can be implemented using an in Algorithm 5.1 and 5.2 for worker and communication server respectively. An eager send returns immediately, since the content of the data is copied over to a packet for transferring. A pair of an arriving message and a matching receive causes two accesses to the hash table by the communication server; one for the arriving message and one by the worker thread for the posted receive. The first of these two operations inserts an entry in the table; the second deletes the entry, copies the data to the receive buffer and frees the packet. If the worker thread comes first, it will yield and will be marked runnable by the communication thread when the receive completes. If it comes second, it will complete the operation immediately.

Our matching mechanism requires to perform at most one operation on each of the three shared data structures. Therefore, the software communication overhead is bounded by a constant as long as these operations take constant time. The following sections describe how to implement these data structure.

We describe now our implementations of the three main data structures used by our library. These custom implementations are specialized to the specific needs of the message-passing library, both in terms of their limited functionality and their adaptation to the specific workload. This customization results in significant performance gains.

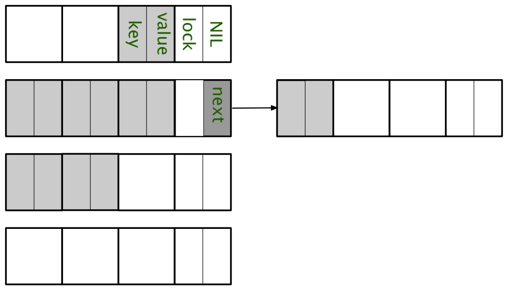


Figure 5.5: An example of our chained hash-table layout with 4 buckets. Each bucket consists of a linked list of 64 byte elements that each contain 3 key-value pairs and a pointer to the next entry.

5.2.3 Concurrent Hash-Table

The shared hash table H stores items that consist of a `<key,value>` pair $< k, v >$. The hash table supports one operation only, defined as follows

$$access(k,v) = \begin{cases} \textbf{if} \ < k, v' > \in H_{pre} \ \textbf{then} \\ \quad H_{post} = H_{pre} - < k, v' >; \ \text{return}(v') \\ \textbf{otherwise} \\ \quad H_{post} = H_{pre} + < k, v >; \ \text{return}(\perp) \end{cases}$$

H_{pre} is the state of the hash table before the access and H_{post} denotes the state after the access.

The hash table has to be *linearizable* [140] which allows the access operation to have *composable* property so that it enables the usage in combination with other concurrent objects. In particular, this ensures that MPI calls take effect in program order.

In particular, when a message has arrived fully, the communication server can perform an access operation with the matching key as tuples of (communicator, rank, tag). If the access returns anything other than $\perp$ which means there is a matching request has been posted by the thread, in this case the request is returned, without inserting anything into the hash table. On the other hand, if there is none, which means no consumer thread has arrived, the communication server can leave the message there (or a pointer to such message) and goes on with other work.

We use a chained hash table with linked lists. The default hash function is the FN-VHash [141] which has been shown to be quite robust and fast compared to other hash functions for 64-bit value [142]. The implementation is optimized for the limited usage we need.

Firstly, we can afford a *spinlock* per bucket, i.e., using an atomic Boolean flag as a ticket to the critical section. This is a viable option since, given no collisions, there are at most two concurrent accesses per bucket by the communication server and a worker. Collisions can be minimized by matching the size of the hash table to the expected number of concurrent communications. A lock-free implementation (using Compare-And-Swap) is possible but is more expensive when the conflicts are rare and results in a more complex code. Thus, we did not pursue this approach.

Secondly, in order to improve cache locality, we design each linked list element as a 4-entry array. Each entry consists of two 64-bit words. One of the entries is used as a *control entry* and the other three are *data entries*. The control entry has the atomic flag for spin locking coupled with a 64-bit pointer to the next element, in case one bucket contains more than three entries. A data entry consists of two 64-bit words of the key-value pair. The total size (64-bytes) thus typically fits in a cache line and one cache miss is the cost of both locking the bucket and fetching the data in the same cache line. Figure 5.5 illustrates an example of our hash-table.

With the above optimization, the `access` operation has close to one cache miss, on average.

5.2.4 Concurrent Packet Pool

In general, a pool structure can be implemented using a lock-free stack. A pool `free` is translated to a stack `push`, and `alloc` is translated to a stack `pop`. At initialization, a fixed number of packets are initialized from the main memory and pushed onto the stack. The Last-In-First-Out (LIFO) property allows good temporal locality for writing/reading to/from the content of a data packet. This design is sufficient for good performance in single-threaded code, but not with multiple cores and in communication settings. Consider a packet recently used by a worker and returned to the pool: this packet could be subsequently obtained by a different worker running on a different core. This causes several cache misses since the cache line is alternately owned by each of the two workers. An example is when two threads running in two different cores alternatively perform `MPI_SEND`.

We explore in this work an implementation which utilized a private local pool for each worker. At runtime, we allow moving packets among those pools via *resource-stealing*, similar to a non-blocking work-stealing algorithm [110]. A private pool is implemented as a fixed size double-ended queue (deque). The deque has three main operations: `popTop`, `pushTop`, and `popBottom` which allows LIFO accessing at one end and removing items from the other end. The packets at the bottom of the deque have been least recently used and are better candidates for use by threads other than the local worker while the other is good for NIC usage since it will lose the locality by being written by the NIC. Thus, we used `popTop` for

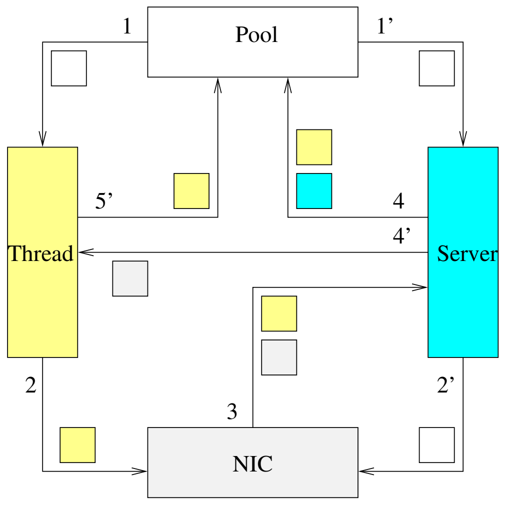


Figure 5.6: Packet life cycle. (1, 2) A worker sending data obtains a packet from the pool, fills the packet and submits to the NIC; (1', 2') Communication server obtains a packet for receiving data and posts the packet to the NIC; (3) The server polls the NIC and obtains the packet back (either sent (yellow) or received (gray) packet). (4) If the packet was for sending, it is returned to the pool (yellow); If the packet was for an incoming send and there is a match, the server copies the data and returns the packet to the pool (blue); (4') If the packet is for an incoming send and the there is no match, the packet is inserted to the hash-table (gray); (5') Following (4') - the worker obtains the packet from the hash-table and copies the data before returning it to the pool (yellow).

a worker thread and `popBottom` by the communication server and for packet stealing (used by other workers). We implemented the pool using a simple ring-buffer and a spinlock.

Figure 5.6 explains how different components in our system might change the affinity of data inside a packet. The figure shows the life of a packet from the time it leaves the pool until it is returned. As explained in the figure, when a packet returns to the pool, it was either last accessed by the communication server or by one of the workers. However, the affinity of a packet to a core is lost once it is posted to the NIC as a receiving buffer since it will be written by the NIC.

Each packet is allocated during initialization with a fixed and configurable size ($64KB$ by default in our implementation). This size corresponds to the maximum size that the eager protocol uses. The number of packets is configurable and typically a multiple of the number of ranks (4 per rank or at least 256 by default in our implementation). Further, to avoid a deadlock situation when all packets are used by the sender, we maintain at all times some available packets for the communication server to pre-post to the network for incoming data. When there is no packet available to the sender, the operation has to be retried at a later time (the LWT yields), which also places a limit on the number of pending requests for congestion and flow control [62].

We expect the pool operations to require at worst 3–5 memory accesses: lock, top, bottom and buffer pointers accesses; a memory read/write for storing a value into the container; and in the rare case of resource stealing, there could be more due to cache misses between processors.

5.2.5 Thread Scheduler

The thread scheduler needs to support the two operations: `ThreadWait` and `ThreadSignal`. We designed a special thread scheduler (*Fult*) to optimize for these two operations. We compare this scheduler to the schedulers in the POSIX Threads library and in Argobots, a system that supports ULTs [121]. Since Argobots implements ULTs, its context-switching mechanism is similar to ours.

In Pthread and Argobots, the `ThreadWait` and `ThreadSignal` operations are implemented using a `condition variable` with a Boolean flag, a generic container for many synchronization primitives. This typically requires a mutex and a queue to store waiting threads. An alternative is to use a busy-waiting synchronization flag that has lower latency; however this is not scalable since the processor spends time polling unnecessarily. In Fult, we use a bit-vector to indicate runnable threads, instead of a queue structure. When a worker is created, it is assigned a unique worker id, denoted as ω. When a LWT starts running on the worker, it is also assigned a unique id Γ. A pair (ω, Γ) uniquely defines a LWT in the system at a point in time. Since ULTs do not migrate, we can maintain a separate bit-vector for each worker; Γ is the index in the bit-vector structure for the bit indicating the status of the corresponding LWT (runnable vs. running/blocked).

The detail analysis and implementation of the FULT scheduler is already described in Chapter 4. FULT was optimized for the two operations, and achieved the lowest possible latency.

5.2.6 Beyond Single Hardware Context

Previous sections and chapters focused on improving concurrent communication with respect to a single hardware context. This allows flexibility in writing applications i.e. all threads can freely communicate without suffering communication degradation. However, this is still being limiting by other resources that are being shared by all threads at the OS kernel layer and below such as DMA engine, or PCIe link. In order to accelerate performance beyond that, we exploit the NIC's capability to provide independent contexts.

We extend LCI with a notion of *device* and *endpoint.* A device represents resources for a hardware context, and a progress entity (e.g. *server*) for making progress for such context. From a device, many endpoints can be created. The endpoint can be thought of as a communicator, with an additional information about a specific protocol that this communicator can handle.

The protocol information is provided at the endpoint creation by a *protocol specification* which determines how an incoming message is handled (*address*) and how completion should be carried (*completion*) when communication is finished. This early binding mechanism eliminates branching in the communication critical path.

The basic communication paradigm that we focus on is that of `producer-consumer`: data are moved from a buffer of the producer to a buffer of the consumer. Completion is triggered at the producer when data is copied out; and at the consumer when the data is available.

LCI defines multiple ways to specify a buffer in a communication call:

- *Piggy-back*: for small data that can be attached to a single packet.
- *Explicit*: a pair of `<address, length>` is specified, which represents a contiguous buffer starting at virtual address `address` and containing `length` bytes.
- *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.

LCI also defines multiple ways to specify a completion event:

- *Completion queue*: Entries providing information on completed events are appended to a *completion queue.*
- *Synchronizer*: A synchronization method that is applied to a synchronization object specified in the call. Synchronization method is compiled with the thread package (with two Signal/Wait calls).
- *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.

Figure 5.7 shows a typical LCI program using one endpoint/hardware per process.

```
// Open the default endpoint, using 1 device
lc_init(1, &ep);

// Create an endpoint option using dynamic allocation, and completion
    queue.
lc_opt opt = {.dev = 0, .desc = {.addr = LC_AR_DYN, .compl = LC_CE_CQ}
    }

// Dup original endpoint using such option.
lc_ep_dup(&opt, ep, &q_ep[0]);
lc_ep_dup(&opt, ep, &q_ep[1]);
...
lc_finalize();
```

Figure 5.7: Example of a LCI program: It creates a default endpoint, which then is used to create two additional endpoints using dynamic allocation for addressing mode and completion queue for completion event.

5.3 COMPONENT ANALYSIS

In this section, we evaluate our implementation of each individual component. This evaluation has several goals.

1. We want to make sure that these components are implemented efficiently and do not suffer from unexpected cache misses.

2. We want to make sure that the total run time is well explained by the run time of the individual components.

3. We demonstrate the performance advantage of using specialized components, which have restricted functionality and are optimized for our use case, as compared to generic components that are commonly used by library developers.

All of our experiments are done on the Stampede cluster and Stampede Knights Landing cluster [45] at TACC. The Stampede cluster (SB) nodes are Intel SandyBridge x86_64 processors with Xeon Dual eight-core sockets, operating at 2.70 GHz, with 32 GB RAM (SB). Each node is equipped with a MT4099 InfiniBand FDR ConnectX interface that is capable of delivering 54 Gbps (mlx). The cluster runs MVAPICH2 MPI version 2.1, compiled with gcc version 4.9.1. The Stampede Knights Landing cluster (KNL) nodes are Intel KNL x86_64 Many Integrated Core processors with 68 cores, operating at 1.4 GHz, with 96 GB of DDR4 RAM and 16GB of Multi-Channel Dynamic Random Access Memory (MCDRAM) configured as L3 cache (Cache-Quadrant) (KNL). They are connected via an Intel Omni-Path fabric - Silicon 100 Series (omni).

All our codes were compiled with mpicc using gcc version 4.9.1 for the SB cluster, and Intel icc version 17-2017.4.196 for the KNL cluster, with `-O3` optimization. We used MVAPICH2 in the SB cluster and Intel MPI in the KNL cluster. Unless noted otherwise, the configuration for these baseline MPI implementations is the default setting with shared memory optimization when running multiple MPI processes per node; when using with POSIX threads, `MPI_THREAD_MULTIPLE` and shared memory optimization are enabled. In both cases, thread affinity is also enabled. We also used PAPI Version 5.5.1.0 on the KNL cluster to perform profiling for some benchmarks.

5.3.1 Communication latency

We evaluate first the communication latency without any of the added overhead due to the runtime implementation of MPI protocol and multi-threaded communication. This measures both the overhead of the low-level communication layer and the transfer time between nodes; it gives us a lower-bound of what we can achieve. For the SB node, we implemented the communication on top of libibverbs, while for the KNL cluster we implemented on top of Libfabrics version 1.3.0. The results are shown in Figure 5.8.

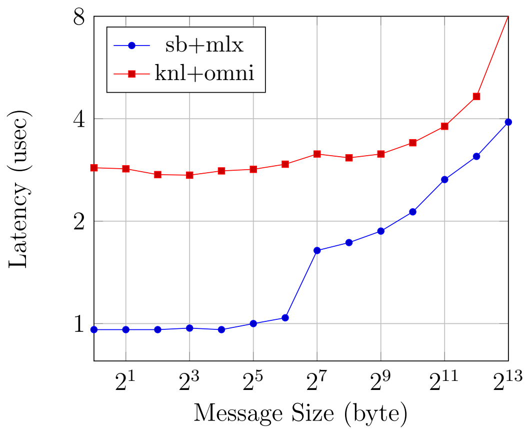


Figure 5.8: Measurement for communication latency lower-bound for various data size for SB cluster (sb+mlx) and KNL cluster (knl+omni). Benchmark is a simple "pingpong" using 2 pre-allocated packets: one send buffer and one receive buffer. In between posting send and receive, a number of network polling operations is required.

Our results show an average latency of **2.8** usec for the KNL cluster and an average latency of **1.0** usec for the SB cluster for moving one cache line of data ($\leq$ 64 bytes). The difference in latency is largely due to the KNL cores being half as fast as the Xeon cores, as they are optimized for throughput rather than latency. Additional differences may be due to the different adapters and communication libraries.

5.3.2 Concurrent Hash-Table

We measure the latency of hash-table operations in the two following scenarios, performing them with multiple POSIX threads:

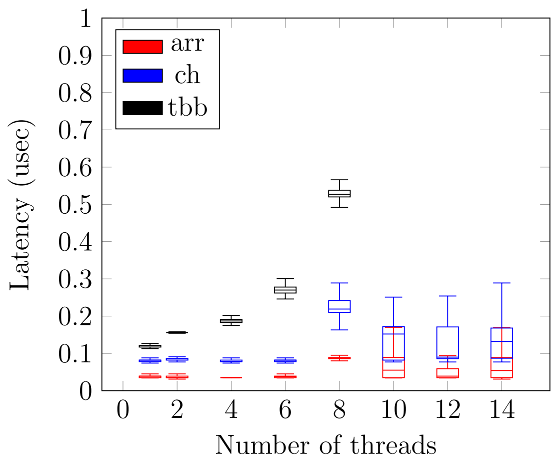
- A thread performs an `access` when there is no item with the same key, which inserts the entry into the hash-table.
- A thread performs an `access` when there is already an item with that key, which deletes the entry from the hash-table.

To justify the benefit of customization, we compare performance to two popular general purposes hash-tables: libcuckoo implementing cuckoo hashing (the spinlock implementation) (*ch*) [143], and TBB concurrent hashmap (*tbb*) [144] – both with default configuration except using the same hash function as our hash table. The version for libcuckoo is the commit 916590 (latest at the time we tested); for TBB is version 2015.2.164 on SB, and 2017.4.196 on KNL. Since there is no native implementation of `access` semantics in both of the hash-table (concurrent hash-table typically provides `insert`, `erase`, `update`, `find`, operations), we emulate that using a combination of `insert` and `erase` (when an entry already exists). We ran the experiments 1000 times, where each time a thread performs 256 operations. Between each run, we also perform a cache invalidation.

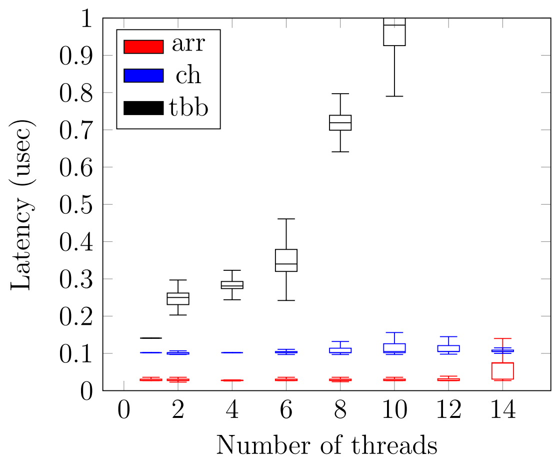
Figure 5.9 and 5.10 show the result of our experiments for latency per operation. Both TBB concurrent hash map and libcuckoo show higher latency and more variance when there are more concurrent threads. The older version of TBB in the SB cluster also performs much worse with threads. Our overhead is almost always as low as **0.05 usec** on the SB cluster and **0.3 usec** on the KNL cluster; the execution time has a low variance and is almost independent of the number of threads.

We attribute these results to the more complex implementation that results in higher overheads and more cache conflicts compared to our simple hash table. For instance, in the KNL cluster, the average number of cache misses (measured with `PAPI_L1_DCM`) is 1.03, 3.05 and 2.37 and the average number of executed instructions (measured with `PAPI_TOT_INS`) per successful access operation is 156, 256 and 456 for arr, libcuckoo, tbb respectively, using 64 threads.

We emphasize that this is not a critique of libcuckoo/tbb; these libraries are more general and optimized for a different regime (few insertions, many searches). These results just indicate the advantage of specialization.

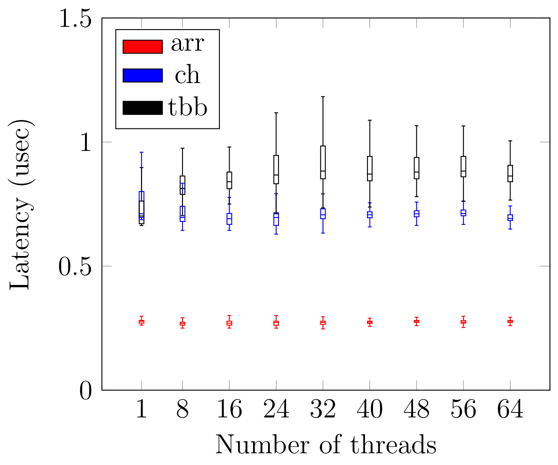


(a) Latency per successful `access` (there is no existing entry).

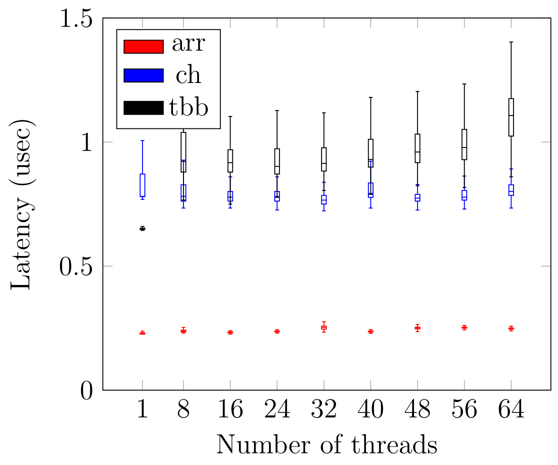


(b) Latency per failed `access` (there is an existing entry).

Figure 5.9: Latency of our hash-table implementation (*arr*) in comparison to libcuckoo (*ch*) and TBB concurrent hash map (*tbb*) for an Intel SandyBridge node. Each hash table is created with the initial size of 2^{16} so that no extension is required during the experiment. TBB is also compiled with *tbb-malloc* to improve performance. Latencies larger than 1 microsecond are not shown.



(a) Latency per successful `access` (there is no existing entry).



(b) Latency per failed `access` (there is an existing entry).

Figure 5.10: Latency of our hash-table implementation (*arr*) in comparison to libcuckoo (*ch*) and TBB concurrent hash map (*tbb*) for an Intel KNL node. Each hash table is created with the initial size of 2^{16} so that no extension is required during the experiment. TBB is also compiled with *tbb-malloc* to improve performance.

5.3.3 Thread scheduler

Table 5.1: Break down of thread scheduler overheads (Fult is our customized scheduler), shown in *usec* for a Intel SandyBridge (SB) and an Intel Knights Landing (KNL). The result is average over 1000 runs, standard deviation is less than 0.02.

Scheduler	**POSIX threads**		**Argobots**		**Fult**	
Processor	SB	KNL	SB	KNL	SB	KNL
Scheduling	0.75	4.5	0.05	0.45	0.02	0.07
Signal	1.15	4.5	0.30	0.85	0.01	0.06
Total	1.90	9.0	0.35	1.30	0.03	0.13

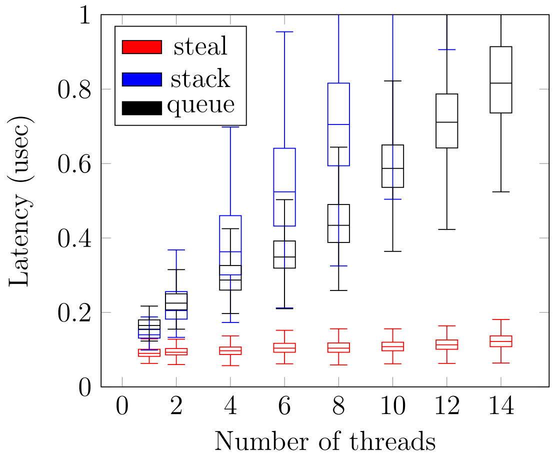
To evaluate the thread scheduler overhead, we measure separately the two operations: 1) How fast can a ULT be scheduled by performing a sequence of yields at a worker and 2) The cost of a `ThreadSignal` by repeatedly issuing the signal on a worker for a ULT at another worker. Table 5.1 shows our results averaged over 1000 runs. Our customized scheduler achieves a total cost of **0.03 usec**, for Signal+Yield, about 10× better than Argobots and 60× better than POSIX threads on SB nodes. Our relative speedup is also similar for KNL nodes, but all of the operations seem to be at least 4× slower compared to SB.

Again, the advantage is due to a much simpler scheduling engine and comes at the expense of the increased generality of a POSIX thread scheduler that supports priorities and signal masks. This generality is not needed by many shared memory programming models.

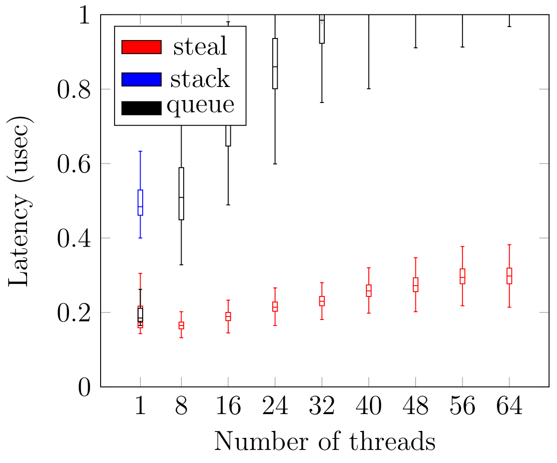
5.3.4 Concurrent Packet Pool

The overhead of packet pool operations is measured as the sum of the latency of an `alloc` and a `free` operation. We evaluate this quantity by performing a random number of `alloc` calls followed by the same number of `free` calls on each thread. To better match with a real workload, we also perform a random sleep in between the two groups of operations. The number of packets allocated per thread is always smaller than the total number of packets divided by the number of threads.

The result is shown in Figure 5.11, in comparison with common implementations using a concurrent lock-free stack and a lock-free queue available in the Boost library [145] (version 1.6). Our result for this benchmark outperforms others by a wide margin, especially when the number of threads increases. The improvement of our data structure comes primarily from the use of thread local storage, which eliminates the memory conflicts presenting in



(a) Latency in a SB node.



(b) Latency in a KNL node.

Figure 5.11: Latency of pool implementation (steal) vs. a lock-free pool (stack) and a lock-free queue (queue) implementation. Latencies higher than 1 microsecond are not shown.

the shared data approaches. Our latency is consistently in the range of **0.1 to 0.15 usec** and **0.2 to 0.3 usec** for SB and KNL node respectively.

Table 5.2: Summary of the lower-bound performance number in *usec* for each component of our MPI communication layer.

	SB + Mellanox	KNL + Omni-Path
Communication (64 bytes)	1.0	2.8
Thread scheduling (fult)	0.03	0.13
Packet Allocation (steal)	0.1	0.2
Message Matching (hashtable)	0.05	0.3
Total	1.18	3.43

5.3.5 Micro-benchmarks

In the previous section, we have analyzed our critical path by evaluating each individual component independently. The overheads are summarized in Table 5.2. Further, our performance results suggest that these overheads should not increase much with an increasing number of workers. In this section, we evaluate the system as a whole to validate the hypothesis using a set of micro-benchmarks.

Table 5.3 shows the different configurations that we evaluate. For MVAPICH2 and Intel MPI, we evaluate single threaded mode (*mvapich2, intel*); single threaded mode with

Table 5.3: Summary of MPI configurations used in the evaluation.

	Msg Matching	**Scheduler**
mvapich2, intel	queues	Single
{mvapich2, intel}+async	queues	Single + Progress
{mvapich2, intel}+mt	queues	POSIX threads
pthread+hash	hash-table	POSIX threads
abt+hash	hash-table	Argobots
fult+hash	hash-table	Fult

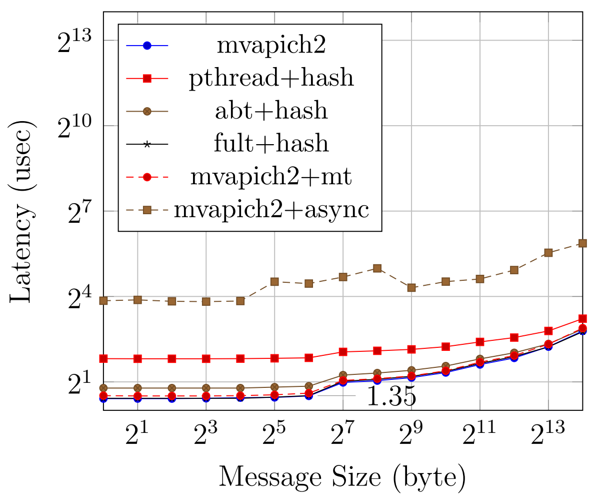
asynchronous progress thread, i.e., `MPICH_ASYNC_PROGRESS`=1 ({*mvapich2, intel*} *+async*); and multi-threaded mode ({*mvapich2, intel*}*+mt*). Using an asynchronous progress thread ({*mvapich2, intel*}*+async*) usually reduces reduces performance quite a lot, as shown in Figure 5.12; the effect is extreme when there is more than one MPI process per node. Therefore, we do not consider this option to be viable, and do not explore it in most of our experiments.

For our implementation, we evaluate three different schedulers: POSIX Threads (*pthread+hash*), Argobots (*abt+hash*) and Fult (*fult+hash*).

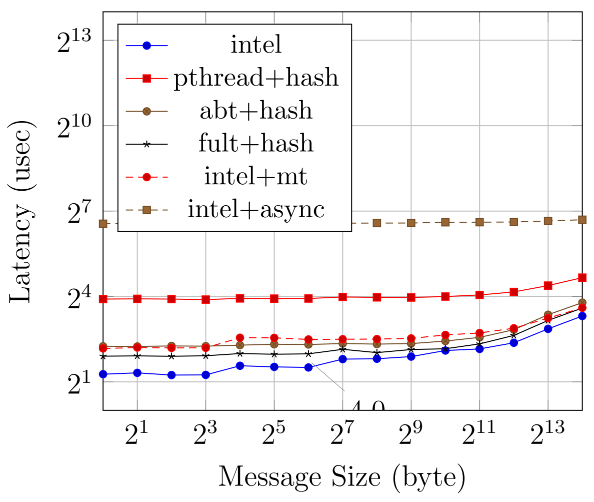
OSU latency benchmarks We use the OSU benchmarks [43] to evaluate the latency per `MPI_SEND` or `MPI_RECV`. The single threaded test is performed using `osu_latency`, the multi-threaded test is performed with `osu_latency_mt`. For a fair comparison in this experiment, we disable the MVAPICH2 *RDMA fast path* algorithms (by setting `MV2_USE_RDMA_FAST_-PATH=0`) . We do not implement this optimization since it relies on an RDMA behavior that is specific to Mellanox devices [146][1]. Further, in multi-threaded tests, we modify the code so that each ULT uses different tags.

OSU singe-threaded latency Performance for the OSU single-threaded test is shown in Figure 5.12. In both clusters our best implementation has lower latency than the default MPI running with `MPI_THREAD_MULTIPLE` (as in *mvapich2+mt* and *intel+mt*). We virtually tie with MVAPICH2 and are less than 1 usec slower than Intel running with `MPI_THREAD_-SINGLE`. The asynchronous progress configuration, even though it sets aside for MPI use the same number of cores as ours, performs much worse in both cases. This shows that optimizing for the single-threaded case still provide some benefit; however the gap in current MPI implementations between `MPI_THREAD_SINGLE` and `MPI_THREAD_MULTIPLE` can be significantly reduced. On both clusters, the use of a progress thread seems unnecessarily onerous.

[1]RDMA data is ensured to be delivered in order (last byte comes last). This only affects the result of single-threaded benchmarks



(a) Latency between 2 nodes in the SB cluster for various message size



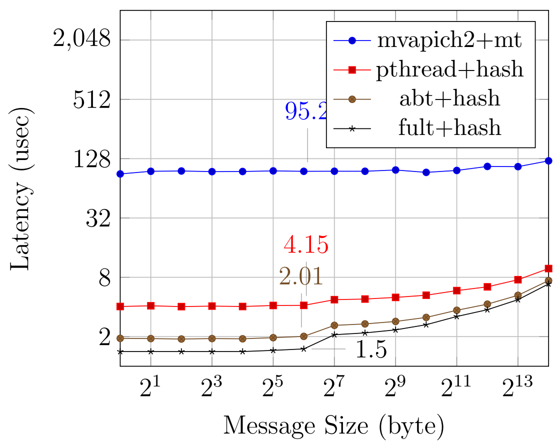
(b) Latency between 2 nodes in the KNL cluster for various message size

Figure 5.12: Latency comparison for single-threaded OSU benchmarks.

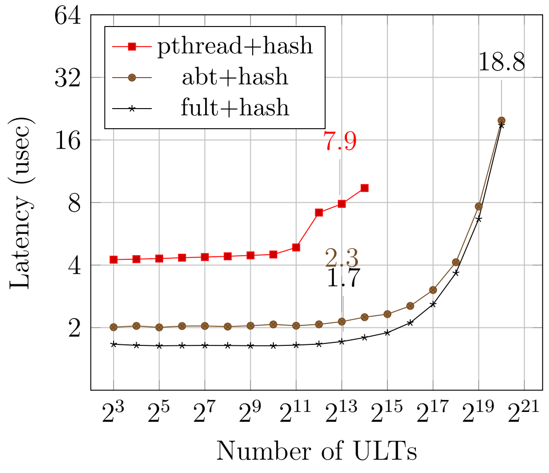
Our performance in both clusters, for messages of 64 bytes, is within **0.1** usec from the totals in Table 5.2. This result confirms our hypothesis that the performance of our software is largely determined by the performance of the three critical components.

OSU multi-threaded latency Performance results for the multi-threaded tests are shown in Figure 5.13. The largest improvement in performance is due to the replacement of the matching queues of MPI with the hash table. Then, the replacement of POSIX threads with user-level threads. Our thread-scheduler improves the latency up to 40% compared to Argobots, 3× compared to POSIX threads scheduler. Overall, we achieve speedup of up to 60× compared to MVAPICH2. The typical communication overhead for MVAPICH2 with a single-threaded process is less than 2 usecs; with 8 threads, the overhead is close to 100 usecs due to synchronization overheads. The relative performance is similar on the KNL cluster, except with higher absolute latency, thus we omit these results from this paper. Our scaling test in Figure 5.13(b) shows that we can support a very large number of threads with very small synchronization overheads. Our performance only degrades slowly at 16K communicating threads, which we attribute to the bottlenecks in memory for thread records (each thread is configured with a 16KB stack for this test).

Since the OSU multi-threaded latency benchmark only increases number of threads in one of the nodes, our results indicate that with blocking MPI, as long as messages are issued at a fixed rate (limited by the single-threaded node in this benchmark), we can support `MPI_THREAD_MULTIPLE` with a very small penalty, if any, compared to `MPI_THREAD_SINGLE`.



(a) Latency per message for 8 threads, one per worker/core.



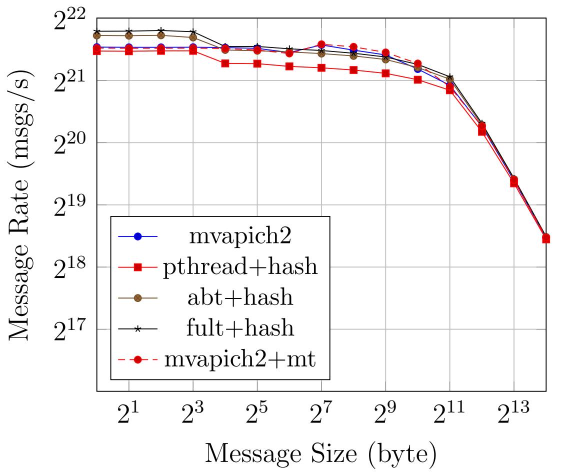
(b) Latency for 64-byte message transfer with up to 1M ULTs that are assigned round-robin to 14 worker/cores, *pthread+hash* version only works up to 16K threads.

Figure 5.13: Latency comparison between different MPI implementation using OSU multi-threaded latency test on the SB cluster.

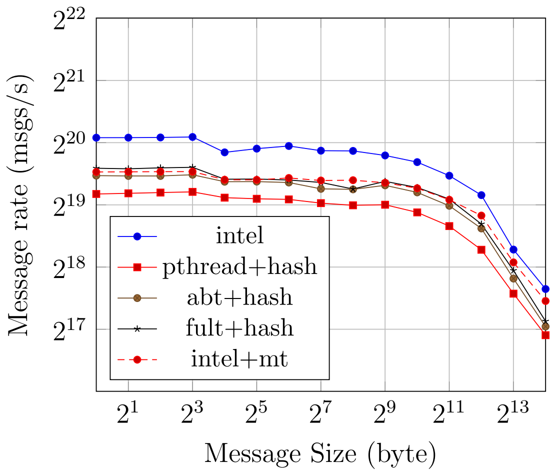
That still holds true even with 16K ULTs. The results clearly indicate that current MPI libraries are optimized for the single threaded case; a different design is needed for supporting concurrent MPI communication from a large number of threads.

OSU bandwidth-message rate benchmark The bandwidth-message rate benchmark (`osu_mbw_mr`) in the OSU benchmark suite can be used to evaluate the implementation of the non-blocking communication. The benchmark creates two ranks; one issues a number of messages (with a *window* parameter defaulted at 64) using `MPI_ISEND`, waits for all to finish with `MPI_WAITALL`, then complete a round of exchange with a `MPI_RECV`. The second MPI rank similarly performs `MPI_IRECV`, `MPI_WAITALL` and `MPI_SEND` in that order to match the first rank. The message rate is computed by taking the number of messages divided by the overall time for the entire experiment. The bandwidth is computed by taking the amount of transferred data divided by the overall time. Since each of these metrics can be derived from the other and we are more interested in the overhead of the communication, we report the message rates and focus on short messages. Similar to the previous section, we compare different MPI implementations and settings as in Table 5.3, except "async".

The results for both SB and KNL cluster are shown in Figure 5.14. The performances of the different implementations do not differ much in the SB cluster: *fult+hash* achieves **3.0** Mmsg/s at 64-bytes, 20% better than *pthread+hash*, 3% better *abt+hash* and 5% better



(a) Performance result in SB cluster for various message sizes



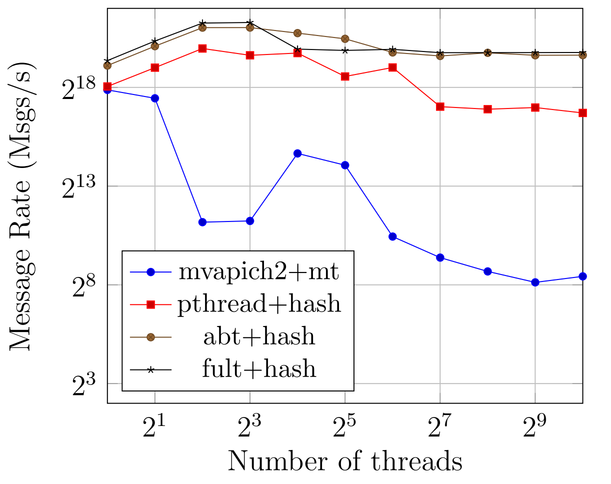
(b) Performance result in KNL cluster for various message sizes

Figure 5.14: Message rate using non-blocking communication comparison using OSU bandwidth-message rate benchmark.

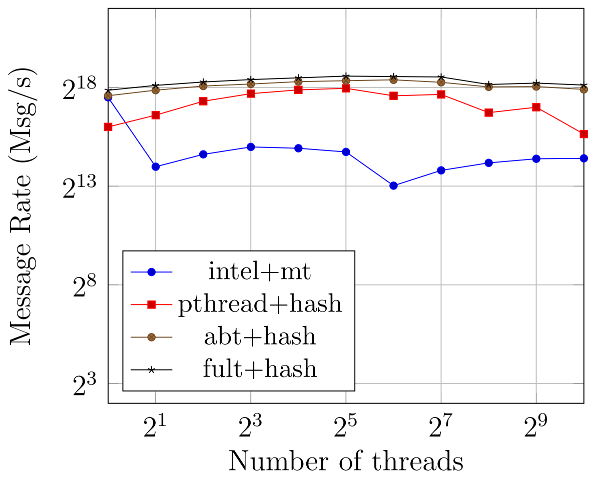
than *mvapich* and *mvapich+mt*. Our performance on the KNL however performs 30% less than *intel*, and 10% less than *intel+mt* for some medium-size messages.

Our implementation requires the communication server to perform operations that are performed by the computing threads in other MPI implementations. The high message rate we can sustain indicates that the communication server on SB is not overcommitted and can handle the network injection rate for 64 byte messages. We do suffer some performance loss in the case of the KNL, where threads are slower, and the communication server handles a larger number of compute cores. The performance loss occurs in the extreme case where all ULTs communicate all the time. The results in the next section show that the communication server is not a bottleneck once the ratio of communication to computation is more reasonable. In any case this performance loss could be mitigated by using a multithreaded communication server and partitioning traffic so that the server threads would need very limited coordination.

New multi-threaded benchmarks Recall that all the benchmarks from the OSU benchmark suite focus on `MPI_THREAD_SINGLE`, except `osu_latency_mt`. This benchmark only increases number of ULTs in one of the two communicating MPI processes. We believe that we can get a more accurate picture by maintaining symmetry across the two ranks. For that purpose, we use a benchmark where each MPI process spawns the same number of ULTs. One ULT on each process will communicate with exactly one ULT on the other process using a distinct tag value. Since "latency" is difficult to define in the present of multiple ULTs, we choose message rate (messages per second) as our metric. The message rate is computed by taking the number of total messages for all ULTs divided by the overall execution times of the benchmark. Each experiment in the section is performed with 10^6 messages equally divided among ULTs.



(a) Performance result in SB cluster using 15 workers.



(b) Performance result in KNL cluster using 64 workers.

Figure 5.15: Multi-threaded message rate benchmark performed in SB and KNL cluster.

Multi-threaded message rate The first benchmark in this section is a simple "ping-pong" between two nodes. Each thread on the first node performs a `MPI_SEND` followed by a `MPI_RECV`, while each thread on the second node performs a `MPI_RECV` followed by a `MPI_SEND`. Each pair of communicating threads is assigned a unique integer value to use as the MPI tag - this makes sure they communicate in pairs. Figure 5.15 presents our results for both SB and KNL cluster using different MPI implementations for this benchmark. In the SB cluster, our MPI using *fult+hash* reaches 2.5 million messages per second (Mmsg/s) then saturates and remains stable at nearly 1 Mmsg/s at 64 or more threads. In the KNL cluster, our *fult+hash* reaches maximum message rate of 0.4 Mmsg/s then saturates and remains at 0.3 Mmsg/s after 256 ULTs. At the point of saturation, *fult+hash* is 10% (8%) better than *abt+hash*, 7× (2×) better than *pthread+hash* and 3000× (15×) better than the default MPI in the SB cluster (KNL cluster, respectively).

We believe that the message rate reaches a peak when multiple packets are returned by each single network poll and the number of threads is small enough to fit all data in cache. Then the performance drops as having more threads sending messages only trashes the cache. This drop appears to be higher in the SB cluster due to the presence of two NUMA nodes, which reduces the performance once many threads span over both. Since a KNL node has more cores, more symmetric layout and a larger L3 cache, it has a later saturation point as well as lesser performance loss. Despite being unable to maintain the peak communication performance with increasingly more communicating ULTs, the result shows that we can still maintain a performance as high as single-threaded communication. The Intel and MVAPICH2 MPI implementations suffer significant synchronization overheads and see their performance drop even with a small number of threads.

In the SB, a peak message rate of 2.5 Mmsg/s corresponds to a *message gap* of 0.4 usec, where the message gap is the delay between two successive message transmissions. This gap is about 1/3 of the total communication latency. In a ping-pong test, each ULT can issue a new send only after a round trip that takes two communication latencies; thus we would expect that a peak message rate be achieved by about 6 ULTs. Similarly in the KNL, the gap is about 2/3 of the latency, which leads to 3 ULTs. This is consistent with our results as seen above.

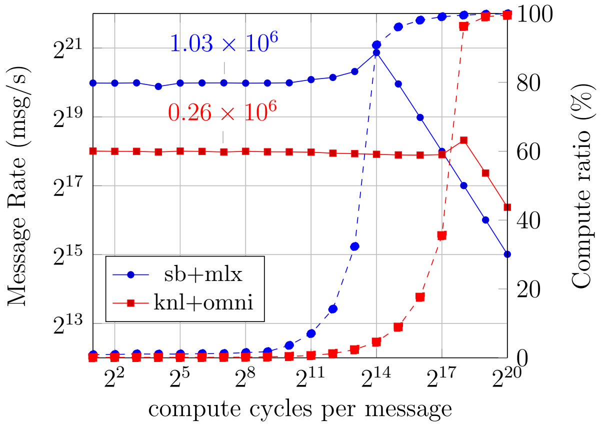


Figure 5.16: Multi-threaded message rate with computation benchmark performed in SB cluster (sb+mlx) and KNL cluster (knl+omni) using our best implementation (fult+hash). The benchmark is performed at the point where the communication is saturated (shown in Figure 5.15 - 64 ULTs, 15 workers in the SB cluster; 256 ULTs, and 64 workers in the KNL cluster). The dotted line of each series corresponds to the ratio of computation over overall time (second y-axis) for each cluster.

Multi-threaded message rate with computation In the previous benchmark each thread only communicates. A more realistic scenario is achieved by adding computation time between communications. This allows us to study the overlap between communication and computation. We modified the previous benchmark to add a fixed number of computation cycles before each `MPI_RECV` and `MPI_SEND`. The message rate is then computed the same as before as total number of messages divided by total execution time.

The benchmark is performed with *fult+hash* in both clusters and the results are shown in Figure 5.16. As we increase the computation per message, the message rate remains flat and even improve slightly up to $16K$ compute cycles for the SB cluster and up to $256K$ compute cycles for the KNL cluster. We believe this indicates several things:

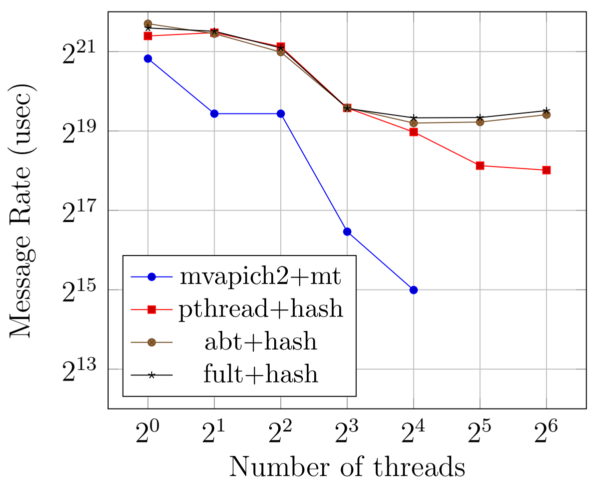
- The message rate per node, for short messages, is limited by the injection rate supported by the NIC and the driving software; this continues to be the limiting factor until we achieve a granularity of $16K$ compute cycles per message for the SB and $256K$ compute cycles per message for the KNL. More frequent communication merely increases idle time, for the chosen number of threads. As shown in Figure 5.16, this corresponds on the SB to an execution where 90% of the time is spent computing and 10% communicating; the numbers for the KNL are 96% and 4%, respectively (These numbers assume 64 byte messages.)

- The message rate per node does not decrease significantly when the compute threads attempt to generate messages at a higher rate than the node can inject in the network: our implementation does not suffer from trashing.

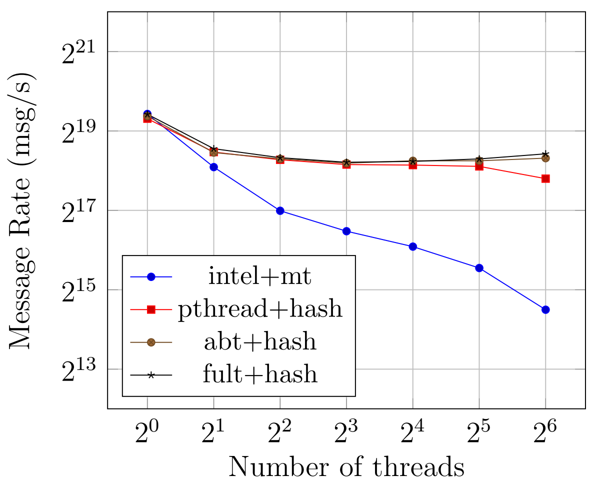
- The message rate reaches a peak when the computation delay of each thread is long enough that the server processing rate is matched, as this also reduces pressure in the memory.

Multi-threaded message rate with non-blocking calls We evaluate here the non-blocking MPI implementation with respect to multi-threaded executions using a new benchmark. This benchmark is modified from our communication-only multi-threaded message rate benchmark to be similar to the `osu_mbw_mr` with a "window" parameter, with an exception that, we use more than one thread in an MPI rank as we did with previous benchmark. Further, as we increase the number of threads in the experiment, we also reduce the *window* parameters proportionally to eliminate the performance effect of increasing number of pending buffers.

The results for both SB and KNL clusters are shown in Figure 5.17. In the SB cluster, there is a drop when threads span over the two NUMA nodes, then the message rate flattens out at **0.7** Mmsg/s; while in the KNL cluster the message rate stays at more than **0.3** Mmsg/s. This result is for *fult+hash*, which is 4% better than *abt+hash* and 20% better than *pthread+hash* on average, using the harmonic mean of the speedups. Both Intel and



(a) Performance result in SB cluster using 15 workers.



(b) Performance result in KNL cluster using 64 workers.

Figure 5.17: Multi-threaded message rate benchmark with non-blocking MPI performed in SB and KNL cluster (from 1 thread, $window = 128$ to 64 threads, $window = 2$).

MVAPICH2 implementations again suffer significant drop in performance when the number of threads increases; compared to the result using blocking calls in Figure 5.15, non-blocking calls do not help when more threads are present, but only worsen the performance.

Communication-computation overlap can be achieved in two ways: Using "heavy" threads and nonblocking communications; or using "light" threads in larger number than physical threads and blocking communications. Current MPI implementations are optimized for the first model. Increased core counts and increased variability in computation and communication time in large systems with dynamic power management favor the second model. Our results show it is possible to implement the latter approach efficiently.

5.4 PERFORMANCE EVALUATION

In this section, we evaluate some mini-applications that represent the kernels of real applications. This gives us a better idea of the runtime performance under real workloads. We performed all of our experiments here using the SB cluster since we did not have access to large numbers of KNL nodes.

We use the NAS *Data Traffic* benchmark to study the impact of different communication patterns on performance. With the *Depth-First-Search* and the *Unbalanced-Tree* benchmarks we compare a conventional implementation using non-blocking MPI calls, with an implementation using message-driven scheduling of ULTs atop our communication library.

5.4.1 NAS Parallel Benchmarks - Data Traffic (DT)

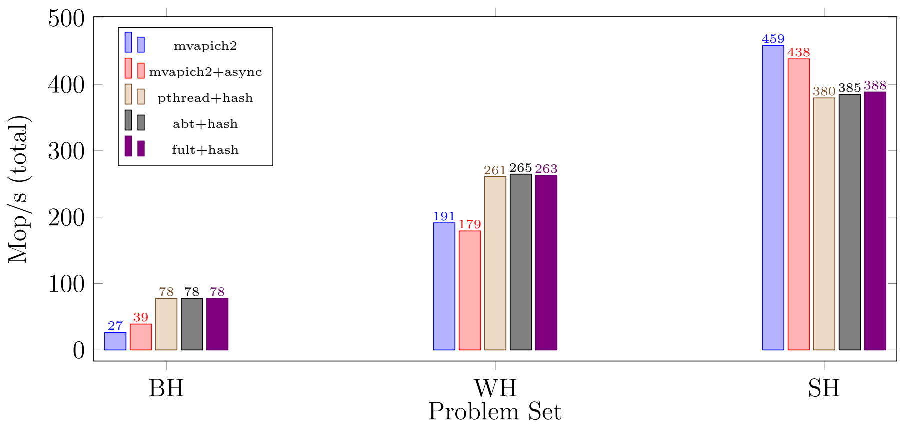


Figure 5.18: Performance of NAS-DT benchmarks in terms of million operations per second (Mop/s - the higher the better) for three different communication patterns under different MPI implementations.

The Data Traffic (DT) code is part of the NAS Parallel Benchmarks. It is used to evaluate the communication performance under three different communication patterns:

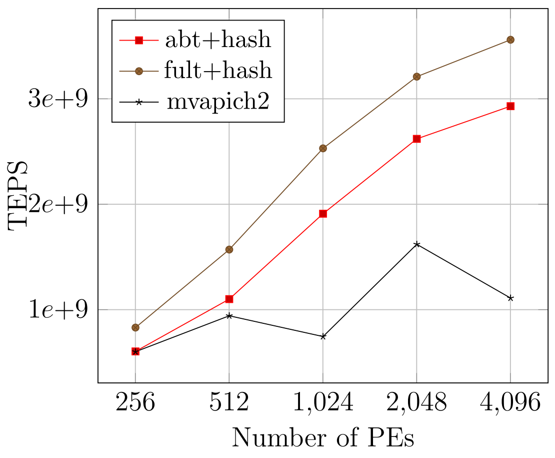
- Black Hole (BH): collects data from multiple sources to a single sink.
- White Hole (WH): distributes data from a single source to multiple sinks.
- Shuffle (SH): routes data from a small number of sources to a small number of sinks through a large numbers of layers.

Between communication phases, there are also significant computations to verify results that help evaluate the ability to overlap communication and computation of the runtime system as well as the effect of cache locality. The application is written with MPI blocking send and receive and each destination rank has a uniquely assigned tag, making it a perfect use case for our MPI implementation. Hence, for this experiment, we execute the reference code using our MPI implementation without changing much of the source nor applying any threading. Since no threading is used, we run the benchmark on 128 nodes, one process per node, two cores per process. Our implementation uses a single worker in comparison with MVAPICH2 in sequential mode and MVAPICH2 with asynchronous progress.

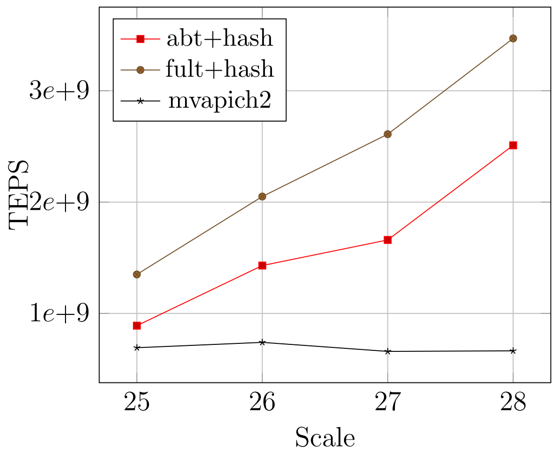
The NBP suite also provides different classes of problem that represent different levels of scale. For the DT benchmark, we evaluate only class "A" since it is reasonably large (requires at least 80 processes), and moreover it is equipped with a proper verification. The reference code was downloaded from NBP suite version 3.3.1 at the NBP website [147].

The results are shown in Figure 5.18. When there is a imbalance in the number of sources and sinks, we perform better in all cases, with up to 3× performance due to a better message matching algorithms. We are about 15% slower in the SH case, due to more cache conflicts. The *mvapich2+async* has lower performance for the same reason, although since the network is polled from both the main thread and the helper threads, it is less affected (average L2 cache misses rate are 23%, 28% and 45% for *mvapich2*, *mvapich2+async* and *fult+hash* respectively as reported by the *perf* profiler).

5.4.2 Breadth-first-search (BFS)



(a) Strong scaling on 29-scale graph.



(b) Weak scaling with 512 PE per 25-scale graph (4096 cores at 28-scale)

Figure 5.19: Strong and Weak scaling for Graph500 in terms of number of Traversed Edges per Second (TEPS - the higher the better) for different MPI implementation.

BFS is the kernel for the *Graph500* benchmark [46], which is frequently used to determine the performance of supercomputers for latency bound applications. The MPI reference implementation generates a large-scale graph and assigns to each MPI process a fixed set of vertices. The implementation then has the processes cooperatively traverse the graph, starting from a particular vertex until all vertices are marked visited. Although the problem is simple, it is often difficult to scale well due to the irregular access pattern and fine-grain communication. The benchmark provides four MPI reference BFS implementations. Among them, `graph500_bfs_simple` is a suitable candidate for us to re-implement since it uses MPI 2-sided point-to-point as the main communication method. Although this implementation has limited scalability, it is also simple to understand and is a frequent target of study. Jose et al. [47] point out that one of the main bottlenecks of this implementation is due to the Send/Recv communication model, which uses non-blocking communication to poll for arriving messages. We attempt to provide a simple remedy for this bottleneck by using blocking Send/Recv in combination with multi-threading. More specifically, we have made the following two important changes:

First, each MPI process traverses its assigned graph partition using multiple processing threads: Each vertex that is assigned to an MPI process is now further assigned to a thread spawned by that process. Each thread also maintains a separate traversing queue and appends to this queue when it traverses vertices that it owns, otherwise, it atomically appends to the queue of the owner threads. For this reason, in our modification, we use all the cores of a compute node within one MPI process, while the reference uses one MPI process per core.

Second, when a vertex is owned by a remote process, the communication is done via blocking `MPI_SEND` and `MPI_RECV` instead of their non-blocking counterpart. A thread performs an `MPI_SEND` when it has accumulated enough vertices owned by the destination. The `MPI_RECV` is performed in a separate set of threads. These threads are spawned initially (assigned to each worker in a round-robin manner), and only scheduled when a message has arrived. The woken up thread finds vertices that belong to the current MPI rank in the receive buffer, and appends them to the corresponding local thread queues; the appends are atomic. The non-blocking receive in the reference implementation uses `MPI_ANY_SOURCE`. Here we apply one of our mitigation strategies by having the number of receiving threads equal to the number of sending nodes.

Although this design could lead to a large number of threads, the receiving threads are not running when there are no incoming messages, hence we do not waste CPU times as in the original algorithm. Moreover, our runtime is able to handle a very large number of threads efficiently as we have shown in the previous section. On the down side, memory for storing thread records may become the bottleneck, in which case one must come up with a more sophisticated approach.

We compare our multi-threaded implementation with 15 workers and 1 communication server (each binds to a processor core) in 1 MPI process with the reference running 16 MPI processes per node; other settings are kept as the default. The weak and strong scaling results of computed median TEPS are provided in Figure 5.19. We do not show the result for *pthread+hash* since the performance is far worse (10× slower than the reference). This is expected since the performance of our threaded version depends on the ability to context-switch efficiently between receiving threads and processing threads, which happens very frequently in BFS due to the fine granularity of the communication. Our implementation using our ULT scheduler is able to scale BFS to 4096 cores. At that scale, our Fult scheduler achieves 3× performance over the reference code and 20% better than Argobots.

5.4.3 Unbalanced Tree Search (UTS)

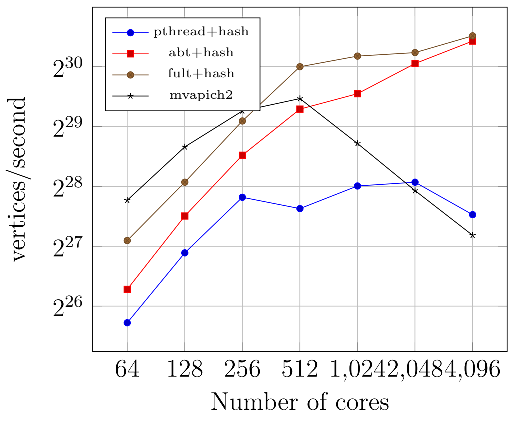


Figure 5.20: Strong scaling results for UTS of T3XXL tree in terms of vertices per second (the higher the better) under different MPI implementations.

Unbalanced Tree Search is a benchmark for evaluating the performance of parallel systems under heavily unbalanced and irregular workloads [115]. The benchmark randomly generates a tree based on sampling from configured probability distribution and requires traversing every generated vertex. Unlike Graph500, an MPI work-stealing implementation is considered quite scalable and has been used for evaluating other runtime systems [56, 148, 149]. The basic idea is that an MPI process sends stealing requests to other MPI processes when it has explored all previously assigned vertices. All communications are done via non-blocking point-to-point MPI calls. The application is less communication bound than the Graph500 BFS, but requires dynamic coordination between the processes for balancing work.

We obtained the latest reference implementation (version 1.1) from the publicly available source at [150] and modified the work-stealing MPI code (`mpi_workstealing.c`) to match our MPI implementation. The modifications are in a similar style as those for Graph500: 1) Use multiple ULTs (assigned to each worker in a round-robin manner) for each MPI process and allow each ULT to explore in parallel multiple vertices in their own stack. When there are no more vertices to explore, the ULT will first try to steal from other threads' stacks on the same node before trying to request work from a different MPI process. We make little effort to optimize the intra-process stealing and use locks to protect critical sections. 2) The reference implementation uses non-blocking communication to wait for work (incoming vertices or incoming stealing requests). Instead, we use multiple communicating threads. Each thread uses a blocking `MPI_RECV` and then acts upon the data it received.

We compare our multi-threaded implementation with 15 workers and 1 communication server (each binds to a processor core) in 1 MPI process with the reference running 16 MPI processes per node; other settings are kept as the default. The strong scaling result is shown in Figure 5.20 for `T3XXL` tree (a 2.8 billion vertices, Binomial tree that is recommended by the package). Our implementation is 10× faster than the original code, 7× than with POSIX threads scheduler and 10% better than with Argobots at 4096 cores.

5.5 SUMMARY AND IMPACT

This chapter describes the motivation, design and implementation of a highly efficient and concurrent message-passing interface. Our method is a tight integration of communication layer, thread scheduler and resource management. Moreover, the entire protocol relies on only a handful of operations which could be potentially implemented in hardware. Using a variety of micro- and application- benchmarks, we proved the efficiency and performance benefit of this design. Our implementation was evaluated thoroughly both on a modern machine such as the Intel SandyBridge with Mellanox and on an emerging platform such as the Intel Knights Landing with Omni-Path. The implementation is able to maintain performance up to a million communicating threads and scale several applications previously shown unscalable, using the same algorithm or with modest modifications.

Our future work will focus on enhancing the functionality of our system and implement other frameworks of different application spaces. While moving bottom-up, we shall evaluate and quantify the cost and benefit of generality vs. performance. We believe the insight from this research will facilitate a discussion on how MPI and communication runtimes should evolve in the upcoming era. The majority of the chapter was published in [151].

CHAPTER 6: CONCLUSIONS AND PROSPECT

This dissertation started by tackling one of the longest standing issue in MPI implementations: issues with multi-threaded communication. For a long time, researcher has considered multi-threaded communication as a bad practice, which does not give any benefit compared to the funneling alternative. And since this model is not performing, none of the application makes use of the model, and so there is no motivation to solve it properly. This vicious cycle has to be taken down, and we have made significant progress towards that goal.

By diving into MPI implementation, we showed that the overhead due to MPI implementation can reduce the communication efficiency by orders of magnitude. These overheads come from: 1) Thread synchronization relies solely on the lock implementation causing unnecessary extra works; 2) The cache misses caused by threads executing in different cores writing to shared resource of the communication runtime; 3) The high overheads of thread synchronization primitives provided by kernel threads such as context switching and signaling.

We have presented methods for solving these problems. Firstly, we designed and Implemented a cooperative scheduling and a thread synchronization method for intelligently reduces the non-useful works of each thread in the communication path. Secondly, we designed and implemented a new lock technique which targets communication frameworks that improves locality and reduces cache misses. Both of these methods have been included in the MPICH 3.3, a widely used MPI implementation.

MPI performance degradation due to the shared-memory multi-core architecture can go beyond multi-threading. Graph analytics framework is one of the emerging users of supercomputing; but it is also affected by because of its communication pattern. The irregular gather-communicate-scatter pattern causes significant performance and memory usage overheads for both traditional MPI one-sided and two-sided patterns. The mismatch in semantic of the MPI eager protocol and the memory usage and performance of for MPI window creation is the root cause for these problems.

We designed a customized communication framework targeting graph analytic requirements, later becomes the foundation of our ultimate communication library. It resolves the performance problem by a simple remote queue interface that avoids unnecessary message-matching while still utilizing RDMA for large messages. The queue interface also improves the memory usage by minimize the amount of buffering compared to MPI one-sided, by reusing the buffer directly through the frameworks memory allocator.

From analyzing frameworks usage of communication library, the notion of fast signal/wait operations come up. These two primitives are the core of producer-consumer synchronizations of many communication patterns. We have formulated the wait-signal problem for message-passing synchronizations, showed the differences from other types of traditional synchronization problems. As a result, we designed and implemented FULT, a specialized thread scheduler targeting communication library. The performance advantage of FULT comes from the novelty of using bit-vectors for implementing thread queues which allows single instruction for waking up a thread.

Putting everything together, we designed and implemented a library targeting highly concurrent message-passing. We relaxed the message-matching semantics of MPI to allow concurrent matching. We take advantages of FULT for reducing overheads of multi-threaded synchronizations to the minimal. Finally, we design the interface to target application-specific use case.

We add to LCI any functionality that can be directly supported by the Network Interface Controller (NIC), to reduce communication latency. For example, LCI provides simple signaling mechanisms to indicate the completion of communication, such as setting a flag, or decrementing a counter. Direct support of such mechanisms by the NIC is easy and can avoid significant polling overheads.

We do not add to LCI any functionality that is more efficiently supported by the higher-level framework. For example, LCI does not provide the equivalent of MPI derived datatype. Frameworks most often use a limited number of data structures and can provide efficient serialization/deserialization routines for those. Packing before sending and unpacking after receiving is almost always more efficient than using a datatype, as the first option uses compiled code, while the second uses an interpreter.

LCI directly supports the communication paradigms that are needed by the frameworks we target, so as to avoid an additional layer that maps framework communication paradigms to the underlying library. For example, many frameworks poll for communication atop MPI, while MPI polls for packets; we wish to avoid this multiple level of polling. Many frameworks use command queue for sending communication requests to the network, LCI allows issuing directly from threads without performance loss.

LCI gives more control to the framework on the allocation of compute resources for communication. Current communication libraries require an asynchronous polling agent to handle incoming messages or various handshakes. Often, these asynchronous activities are performed as a side effect of communication calls. As a result, communication class can have widely varying execution time. The framework is better placed to know when more resources should be allocated to communication or computation. The issue plagues libraries using the active message paradigm.

LCI provides more control to the framework on the allocation of memory. Current libraries, such as MPI, use a rigid – static allocation of memory for communication buffering. This is detrimental to applications where communication volume and topology varies widely, as is the case for graph analytics. A too small allocation may cause the application to crash; this does happen when MPI send-receive uses the eager protocol and sends run ahead of receives. A safe allocation will overuse buffer memory. One reason for this problem is that there is no push-back to the application when communication using the eager protocol over-commits buffer space.

LCI allows efficiently large numbers of concurrent threads and of concurrent communications. To the extent possible, LCI directly communicates data between a producer thread and a consumer thread, avoiding the need for additional communication agent at the framework or application level.

The changes in computer architecture during the last decade have shifted the overheads towards the software stack significantly. Multi-core clusters with mixed-in accelerators complicates how data is moved between devices. LCI provides a simple and generic interface which is easy to integrate with any higher-level library. This allows any entity (threads, tasks, or accelerator) to make communication directly to the network, without extra overheads for maintaining thread-safety and correctness.

The APIs of LCI is still actively under-development. We continue to evolve LCI for using in other frameworks in the near future. The source code of LCI can be found in `http://github.com/danghvu/LCI`.

REFERENCES

[1] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Communication-optimal parallel and sequential cholesky decomposition," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3495–3523, 2010.

[2] M. Snir, "A note on n-body computations with cutoffs," *Theory of Computing Systems*, vol. 37, no. 2, pp. 295–318, 2004.

[3] M. Adler, W. Dittrich, B. Juurlink, M. Kutyłowski, and I. Rieping, "Communication-optimal parallel minimum spanning tree algorithms," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures.* ACM, 1998, pp. 27–36.

[4] J. Chen, H. Sun, D. Woodruff, and Q. Zhang, "Communication-optimal distributed clustering," in *Advances in Neural Information Processing Systems*, 2016, pp. 3727–3735.

[5] J. Demmel, "Communication-avoiding algorithms for linear algebra and beyond." in *IPDPS*, 2013, p. 585.

[6] M. Hoemmen, "Communication-avoiding krylov subspace methods," Ph.D. dissertation, UC Berkeley, 2010.

[7] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, "Ca-svm: Communication-avoiding support vector machines on distributed systems," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.* IEEE, 2015, pp. 847–859.

[8] M. Snir, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: The MPI core.* MIT press, 1998, vol. 1.

[9] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[10] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 4, pp. 273–298, 1986.

[11] E. W. Dijkstra, "The structure of the the multiprogramming system," in *The origin of concurrent programming.* Springer, 1968, pp. 139–152.

[12] P.-J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with readers and writers," *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[13] C. A. R. Hoare, "Monitors: An operating system structuring concept," in *The origin of concurrent programming.* Springer, 1974, pp. 272–294.

[14] W. Gropp and M. Snir, "Programming for exascale computers," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 27–35, 2013.

[15] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[16] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem," *Parallel Computing*, vol. 33, no. 9, pp. 634–644, 2007.

[17] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid mpi: efficient message passing for multi-core systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM, 2013, p. 18.

[18] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive mpi," in *International workshop on languages and compilers for parallel computing.* Springer, 2003, pp. 306–322.

[19] H. Tang and T. Yang, "Optimizing threaded mpi execution on smp clusters," in *Proceedings of the 15th international conference on Supercomputing.* ACM, 2001, pp. 381–392.

[20] M. Pérache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting.* Springer, 2009, pp. 94–103.

[21] A. Denis, "pioman: a pthread-based multithreaded communication engine," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on.* IEEE, 2015, pp. 155–162.

[22] N. Hjelm, M. G. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving mpi multi-threaded rma communication performance," in *Proceedings of the 47th International Conference on Parallel Processing.* ACM, 2018, p. 58.

[23] S. J. Plimpton and K. D. Devine, "Mapreduce in mpi for large-scale graph algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, 2011.

[24] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the mpi standard," *Communications of the ACM*, p. 18, 1995.

[25] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2. ACM, 1992, pp. 256–266.

[26] D. Bonachea and P. Hargrove, "Gasnet specification, v1. 8.1," 2017.

[27] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[28] D. Grünewald and C. Simmendinger, "The gaspi api specification and its implementation gpi 2.0," in *7th International Conference on PGAS Programming Models*, 2013, p. 243.

[29] "OFA," https://ofiwg.github.io/libfabric/.

[30] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss et al., "Ucx: an open source framework for hpc network apis and beyond," in *HOTI*. IEEE, 2015, pp. 40–43.

[31] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *EuroMPI '08*, pp. 120–129.

[32] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded MPI communication on multicore petascale systems," in *EuroMPI '10*, pp. 11–20.

[33] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," in *PPoPP '12*, pp. 247–256.

[34] M. Chabbi and J. Mellor-Crummey, "Contention-conscious, locality-preserving locks," in *PPoPP '16*, pp. 22:1–22:14.

[35] D. Dice, "Malthusian locks," *CoRR*, vol. abs/1511.06035, 2015.

[36] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+threads: Runtime contention and remedies," in *PPoPP '15*, pp. 239–248.

[37] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[38] "Intel MPI Library," https://software.intel.com/en-us/intel-mpi-library/.

[39] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda, "Design of high performance MVAPICH2: MPI2 over InfiniBand," in *CCGrid '06*, pp. 43–48.

[40] A. Amer, H. Lu, Y. Wei, J. Hammond, S. Matsuoka, and P. Balaji, "Locking aspects in multithreaded MPI implementations," Argonne National Lab., Tech. Rep. P6005-0516, 2016.

[41] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[42] T. Craig, "Building FIFO and priority-queuing spin locks from atomic swap," University of Washington, Tech. Rep. TR 93-02-02, 1993.

[43] P. Dhabaleswar, "OSU Micro-Benchmarks 5.3," http://mvapich.cse.ohio-state.edu/benchmarks/, 2016, [Online; accessed 18-April-2016].

[44] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[45] "TACC Stampede Cluster," http://www.xsede.org/resources/overview, 2016.

[46] "Graph 500," http://www.graph500.org/, [Online; accessed 13-May-2016].

[47] J. Jose, S. Potluri, K. Tomko, and D. K. Panda, "Designing scalable Graph500 benchmark with hybrid MPI+OpenSHMEM programming models," in *Supercomputing.* Springer, 2013, pp. 109–124.

[48] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[49] F. García, A. Calderón, and J. Carretero, "MiMPI: A multithread-safe implementation of MPI," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting.* Springer, 1999, pp. 207–214.

[50] A. Skjellum, B. Protopopov, and S. Hebert, "A thread taxonomy for MPI," in *MPI Developer's Conference, 1996. Proceedings., Second.* IEEE, 1996, pp. 50–57.

[51] H. Tang, K. Shen, and T. Yang, "Compile/run-time support for threaded MPI execution on multiprogrammed shared memory machines," in *ACM SIGPLAN Notices*, vol. 34, no. 8. ACM, 1999, pp. 107–118.

[52] E. D. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *Proceedings of the 11th international symposium on high performance computing systems.* Citeseer, 1997, pp. 153–163.

[53] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, and C. L. Mendes, "A new technique for data privatization in user-level threads and its use in parallel applications," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 2149–2154.

[54] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Euro-Par '08*, pp. 78–88.

[55] O. Aumage, G. Mercier, and R. Namyst, "MPICH/Madeleine: a true multi-protocol MPI for high performance networks," in *Parallel and Distributed Processing Symposium., Proceedings 15th International.* IEEE, 2001, pp. 8–pp.

[56] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), 2013.* IEEE, 2013, pp. 712–725.

[57] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, "Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications," in *Proceedings of the 2014 Workshop on Exascale MPI.* IEEE Press, 2014, pp. 9–19.

[58] H. Lu, S. Seo, and P. Balaji, "MPI+ ULT: Overlapping communication and computation with user-level threads," in *2015 IEEE 17th International Conference onHigh Performance Computing and Communications (HPCC).* IEEE, 2015, pp. 444–454.

[59] S. G. Caglar, G. D. Benson, Q. Huang, and C.-W. Chu, "USFMPI: a multi-threaded implementation of MPI for Linux clusters," in *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, 2003, pp. 674–680.

[60] 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.

[61] A. Denis, "pioman: a pthread-based multithreaded communication engine," in *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).* IEEE, 2015, pp. 155–162.

[62] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu, "Congestion avoidance on manycore high performance computing systems," in *Proceedings of the 26th ACM international conference on Supercomputing.* ACM, 2012, pp. 121–132.

[63] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *OOPSLA '93*, pp. 91–108.

[64] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *SC '15.*

[65] H. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *EuroMPI '16.*

[66] 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.* IEEE Press, 2017, pp. 314–324.

[67] Programming Models and Runtime Systems, Argonne National Laboratory, "MPICH 3.3 Release," https://github.com/pmodels/mpich/blob/master/CHANGES.

[68] O. MPI, "Open MPI faq," https://www.open-mpi.org/faq/?category=all# thread-support.

[69] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 17–30.

[70] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:15.

[71] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD Intl Conf. on Management of Data*, ser. SIGMOD '10, 2010, pp. 135–146.

[72] "Apache Giraph," http://giraph.apache.org/, 2013.

[73] A. Buluc and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[74] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *GRADES*, 2013.

[75] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *OSDI*, 2016, pp. 301–316.

[76] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 291–305.

[77] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "PGX.D: A fast distributed graph processing engine," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 58:1–58:12.

[78] F. Yang, M. Wu, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "GraM: Scaling graph computation to the trillions," in *SoCC*. ACM, August 2015.

[79] L. Lu, "The diameter of random massive graphs," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000, pp. 912–921.

[80] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, no. 5, pp. 78–87, Apr. 2016.

[81] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The TAO of parallelism in algorithms," in *PLDI*, 2011, pp. 12–25.

[82] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *Proceedings of the 23rd European MPI Users' Group Meeting.* ACM, 2016, pp. 1–14.

[83] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, "Abelian: A compiler and runtime for graph analytics on distributed, heterogeneous platforms," in *To appear in Euro-Par 2018*, 2018.

[84] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[85] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *European Conference on Parallel Processing.* Springer Berlin Heidelberg, 2015, pp. 438–450.

[86] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine, "Fixing probe for multi-threaded mpi applications (revision 4)," Technical report, Indiana University (January 2009), Tech. Rep., 2009.

[87] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. De Supinski, and A. Lumsdaine, "Efficient MPI support for Advanced Hybrid Programming Models." *EuroMPI*, vol. 10, pp. 50–61, 2010.

[88] M. Li, X. Lu, S. Potluri, K. Hamidouche, J. Jose, K. Tomko, and D. K. Panda, "Scalable graph500 design with MPI-3 RMA," in *CLUSTER*, Sept 2014, pp. 230–238.

[89] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the mpi 3.0 one-sided communication interface," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 17, pp. 4385–4404, 2016.

[90] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss et al., "UCX: An open source framework for HPC network APIs and beyond," in *HotI '15*, pp. 40–43.

[91] "Openfabrics interfaces," https://ofiwg.github.io/libfabric, online; accessed 31 March 2017.

[92] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," *SIGPLAN Not.*, vol. 48, no. 8, pp. 103–112, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442527

[93] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013.

[94] J. Nieplocha and B. Carpenter, "Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems," *Parallel and Distributed Processing*, pp. 533–546, 1999.

[95] D. Bonachea, "Gasnet specification, v1. 1," University of California at Berkeley, Tech. Rep. CSD02-1207, 2002.

[96] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999.

[97] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques.* ACM, 2010, pp. 401–410.

[98] M. Besta and T. Hoefler, "Accelerating irregular computations with hardware transactional memory and active messages," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing.* ACM, 2015, pp. 161–172.

[99] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming processing in the network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 59:1–59:16.

[100] H.-V. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, and M. Snir, "A lightweight communication runtime for distributed graph analytics," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2018, pp. 980–989.

[101] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, 2018.

[102] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[103] L. V. Kale and S. Krishnan, *CHARM++: a portable concurrent object oriented system based on C++.* ACM, 1993, vol. 28, no. 10.

[104] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Supercomputing 2012.* IEEE Computer Society Press, 2012, p. 66.

[105] R. Bagrodia, R. Meyer, M. Takai, Y.-a. Chen, X. Zeng, J. Martin, and H. Y. Song, "Parsec: A parallel simulation environment for complex systems," *Computer*, vol. 31, no. 10, pp. 77–85, 1998.

[106] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 202–229, 1998.

[107] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[108] Wikipedia, “Spurious wakeup,” https://en.wikipedia.org/wiki/Spurious_wakeup.

[109] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[110] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” *Theory of Computing Systems*, vol. 34, no. 2, pp. 115–144, 2001.

[111] D. Hendler and N. Shavit, “Non-blocking steal-half work queues,” in *Proceedings of the twenty-first annual symposium on Principles of distributed computing.* ACM, 2002, pp. 280–289.

[112] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, “A simple load balancing scheme for task allocation in parallel achines,” in *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA ’91. New York, NY, USA: ACM, 1991, pp. 237–245.

[113] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, “Using PAPI for hardware performance monitoring on Linux systems,” in *Conference on Linux Clusters: The HPC Revolution*, vol. 5. Linux Clusters Institute, 2001.

[114] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Peña, “Glt: A unified api for lightweight thread libraries,” in *European Conference on Parallel Processing.* Springer, 2017, pp. 470–481.

[115] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, “UTS: An unbalanced tree search benchmark,” in *Languages and Compilers for Parallel Computing.* Springer, 2006, pp. 235–250.

[116] Microsoft, “MSDN Library: Fibers,” https://msdn.microsoft.com/en-us/library/ms682661.aspx.

[117] Oracle, “Programming with Solaris Threads,” https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033n/index.html.

[118] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, “Converse: An interoperable framework for parallel programming,” in *Parallel Processing Symposium, 1996., Proceedings of IPPS’96, The 10th International.* IEEE, 1996, pp. 212–217.

[119] A. Porterfield, N. Nassar, and R. Fowler, “Multi-threaded library for many-core systems,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 1–8.

[120] J. Nakashima and K. Taura, "Massi: A thread library for high productivity languages," in *Concurrent Objects and Beyond*. Springer, 2014, pp. 222–238.

[121] S. Seo, A. Amer, P. Balaji, P. Beckman, C. Bordage, G. Bosilca, A. Brooks, A. Castell, D. Genet, T. Herault, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, and Y. Sun, "Argobots: A lightweight low-level threading/tasking framework," Argonne National Laboratory, Tech. Rep. ANL/MCS-P5515-0116, 2016.

[122] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IPDPS 2008*. IEEE, 2008, pp. 1–8.

[123] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Pena, "GLTO: On the Adequacy of Lightweight Thread Approaches forOpenMP Implementations," in *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 2017, pp. 60–69.

[124] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain, "The Chapel tasking layer over Qthreads," Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2011.

[125] S. Perarnau, R. Thakur, K. Iskra, K. Raffenetti, F. Cappello, R. Gupta, P. Beckman, M. Snir, H. Hoffmann, M. Schulz et al., "Distributed monitoring and management of exascale systems in the Argo project," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2015, pp. 173–178.

[126] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *European MPI group meeting (EuroMPI) 2016*, 2016.

[127] A. Castelló, A. J. Pena, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Ortí, "A review of lightweight thread approaches for high performance computing," in *Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 471–480.

[128] H.-V. Dang and M. Snir, "Fult: Fast user-level thread scheduling using bit-vectors," in *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018, p. 71.

[129] K. Bergman, S. Borkar, D. Campbell et al., "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[130] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," in *Journal of Physics: Conference Series*, vol. 180. IOP Publishing, 2009, p. 012045.

[131] UPC Consortium, "UPC language specifications v1.3," http://upc.lbl.gov/publications/upc-spec-1.3.pdf, 2013.

[132] P. Charles, C. Grothoff, V. Saraswat et al., "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPlan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[133] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *ACM Sigplan Notices*, vol. 33. ACM, 1998, pp. 212–223.

[134] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 2.

[135] T. G. Mattson, R. Cledat, V. Cav, V. Sarkar, Z. Budimli, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, "The Open Community Runtime: A runtime system for extreme scale computing," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–7.

[136] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex: an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 394–401.

[137] A. Brooks, H.-V. Dang, N. Dryden, and M. Snir, "PPL: an abstract runtime system for hybrid parallel programming," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM, 2015, pp. 2–9.

[138] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *International Supercomputing Conference*, 2016.

[139] Message Passing Interface Forum, "MPI 4.0 standardization effort, point to point communication," https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/PtpWikiPage, [Online; accessed 6-May-2016].

[140] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[141] "FNV hash," http://isthe.com/chongo/tech/comp/fnv/, [Online; accessed 20-June-2017].

[142] J. Karasek, R. Burget, and O. Morskỳ, "Towards an automatic design of non-cryptographic hash function," in *Telecommunications and Signal Processing (TSP), 2011 34th International Conference on*. IEEE, 2011, pp. 19–23.

[143] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 27.

[144] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[145] "Lock-free stack and queue," http://www.boost.org/doc/libs/1_60_0/doc/html/boost/lockfree/, 2016, [Online; accessed 2-May-2017].

[146] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[147] NASA, "NAS Parallel Benchmarks," http://www.nas.nasa.gov/publications/npb.html, 2016.

[148] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2007, pp. 1–8.

[149] R. Machado, C. Lojewski, S. Abreu, and F.-J. Pfreundt, "Unbalanced tree search on a manycore system using the GPI programming model," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 229–236, 2011.

[150] "The unbalanced tree search benchmark," https://sourceforge.net/projects/uts-benchmark/files/, 2016.

[151] H.-V. Dang, M. Snir, and W. Gropp, "Eliminating contention bottlenecks in multi-threaded MPI," *Parallel Computing*, vol. 69, pp. 1–23, 2017.