Eliminating contention bottlenecks in multithreaded MPI

Hoang-Vu Dang, Marc Snir, William Gropp

Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Urbana, IL 61801, USA

A R T I C L E   I N F O

Article history:
Received 16 January 2017
Revised 30 June 2017
Accepted 17 August 2017
Available online 18 August 2017

M S C:
00-01
99-00

Keywords:
MPI
User-level threads
Communication
Networking
Scheduling

A B S T R A C T

We explore in this paper the advantages that accrue from avoiding the use of wildcards in MPI. We show that, with this change, one can efficiently support millions of concurrently communicating light-weight threads using send-receive communication. This is achieved through a streamlined implementation of MPI and a tight coupling between the communication runtime and the thread scheduler.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

The first version of the Message Passing Interface (MPI) was designed more than twenty years ago [1]. The design was influenced by the existing message-passing formalisms, such as CSP [2], the practice of then extant message-passing libraries, and the needs of scalable parallel computers at the time. This resulted in fairly complex rules for matching sends to receives: MPI matches sends to receives using values for communicator, sender rank and tag; the receive operation can specify a wildcard value for sender rank and tag. Sends and receives must be matched in order: If a receive posted at A can match two distinct sends from B, then the receive will be paired with the older send; and if a send matches two distinct receives, it will be paired with the older receive. Finally, the implementation must handle sends that occur before a matching receive is posted, as well as those that occur in the reverse order.

Since communication was relatively slow, the additional software overhead of a more complex protocol was not a significant issue. This was balanced against the advantage of supporting the practice of different message-passing libraries. Most importantly, MPI-1 targeted single threaded processes, since computers at the time used single core processors. Explicit MPI support for multi-threaded processes appeared only in MPI-2 [3]. Not all early MPI implementations were thread-safe, so that there was little use of MPI_THREAD_MULTIPLE and little concern for the performance of multithreaded MPI.

The situation has changed a lot in the last twenty years: Computer nodes can now have hundreds of concurrent hardware threads. Modern adapters can support a significant fraction of the communication stack in hardware or firmware, and could support all the performance-critical components if the protocol was streamlined. Energy-efficient, throughput-oriented cores
execute the MPI library code more slowly. Programmers have shifted to the use of hybrid parallelism: Shared memory parallelism, e.g., with OpenMP for intranode, and message-passing parallelism for internode. However, in almost all applications, MPI calls are executed only in sequential sections of the OpenMP code, and MPI is used in tunneled mode, with all MPI calls being executed by one thread. Indeed, we could not find significant application codes that use MPI_THREAD_MULTIPLE. This constraint results in less efficient OpenMP code with more frequent serialization points.

One reason for this state of affairs is that the support for multi-threaded MPI is still imperfect. The latest version of the Open MPI implementation of MPI still does not provide stable support of MPI_THREAD_MULTIPLE [4]. Several studies have reported significant overheads for performing MPI calls concurrently on many threads [5,6]. This is, in part, an engineering issue: MPICH provided thread-safety via a coarse-grain lock which essentially serialized all MPI code. The research to replace coarse-grain locking is still ongoing [6–8], but progress is slow because the support for multi-threading comes at the expense of single thread performance (which is what benchmarks measure). For example, there are a significant number of global objects shared by the executing threads. Global objects that implement stacks, queues or hash tables can be replaced with concurrent data structures, and others might be protected with finer-grain locks. However, finer grain locks and concurrent data structures have higher overheads that add to the critical path length and reduce the performance of single-threaded applications.

The semantics of matching sends to receives complicates the design of efficient support for MPI_THREAD_MULTIPLE. The simplest 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 list. 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. 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: 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 [9], 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 [10].

Another impediment to the efficient support of MPI_THREAD_MULTIPLE is the lack of integration between the MPI library and the thread library. Consider a thread that executes a blocking MPI call, such as MPI_RECV or MPI_WAIT. The thread may yield and should be rescheduled when an appropriate communication completes. The MPI library can track, for each blocked thread, which communication event makes the thread runnable again. However, there is no simple way of passing this information to the scheduler. The thread library will wake up threads irrespective of the status of the communication they are waiting for; the thread will check some flag and yield again, if the communication is not complete. This is not a problem if most of the time only one thread is blocked — in particular, if only one thread executes MPI calls; but can be very inefficient if the number of blocked threads is large. If a user thread library is used (as is customary in parallel computing), then one needs an efficient mechanism for marking a specific thread as runnable when a blocking communication has completed. In addition, we need dedicated cores or proper interaction between the kernel thread scheduler and the user thread scheduler, e.g., using Scheduler activation techniques [11], in order to ensure that threads are not descheduled by the kernel while holding locks.

MPI needs to evolve so as to support efficient concurrent communication from a large number of threads. This can be done by a combination of changes in the MPI implementation and relaxations of the MPI semantics. The evolution needs to be guided by a clear understanding of the trade-offs involved. This paper presents a step in this direction. We study how efficiently we can support send-receive with more limited MPI send-receive semantics. This can be used both to provide a more efficient send-receive layer to applications that can accept the restrictions we impose, and to quantify the cost of supporting richer semantics. This bottom-up strategy provides insight on the minimal number of cache misses that are necessary in send-receive communication. The protocol we propose is simple enough to be implemented in hardware.

The main result of this paper is that, if wildcard receives are avoided, the performance difference between MPI_THREAD_SINGLE and MPI_THREAD_MULTIPLE largely disappears. Furthermore, communication performance does not deteriorate, even with many thousands of concurrently communicating threads.

The main ingredients of our efficient message-passing runtime are

- A light-weight thread scheduler using a bit-vector that requires a single write for marking a thread as runnable.
- A constant time overhead algorithm for MPI point-to-point communication utilizing a specially designed concurrent hash-table.
- A resource-aware and locality-aware concurrent memory pool for packet management.
- An MPI runtime design for scaling millions of communicating threads.

This paper is an extended version of a conference paper [12]. We added a discussion of non-blocking calls and provided new performance benchmarks to evaluate our implementation. We also ported our software to Libfabrics - a portable Open
Our paper is organized as follows. The next section describes the simplified send-receive model that we target and introduces our runtime design. We present our implementation details and optimization in Section 3. Section 4 discusses experiment and results. Section 5 gives an overview of some related work. Section 6 addresses some of the unanswered questions in our work. Finally Section 7 concludes our study and discusses future work.

2. Runtime architecture

2.1. Restricted send-receive model

We focus in this paper on the exchange of messages that are sent from and received into contiguous buffers with matching sizes; the handling of datatypes would be done by a layer atop this basic communication layer. We assume that sends and receives are matched using a key $k$ (in MPI, this is the $\langle$communicator, sender, tag$\rangle$ triplet), and do not support wildcards. Furthermore, we assume that each send can be matched by exactly one receive, so that ordering is a moot issue. We shall discuss in Section 6 how we can relax these assumptions; the only important one is the prohibition of wildcard receives.

We focus on performance-critical operations, namely sends and receives. We ignore, for the time being, one-sided and collective communications.

We assume a light-weight thread library, where thread scheduling does not involve the kernel.

2.2. High-level system design

We propose an implementation of message-passing 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 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.

Fig. 1 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.

As we will show in Section 4, one single-threaded communication server can support the communication traffic generated by all other threads on a node, so our current implementation uses a single-threaded communication server. We plan to explore in future work the use of multi-threaded communication servers.
The workers execute User Level Threads (ULT) (aka tasks). These are managed by a ULT scheduler. The scheduler is simply a function invoked when a ULT completes or yields. A ULT executing a blocking MPI call will yield, and invoke the ULT scheduler; the scheduler will schedule another ULT on that worker. We assume in this paper that the association of ULTs to workers is fixed: ULTs are not migrated once they started executing. We discuss in Section 6 how one can remove this constraint.

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 ULT 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 (\texttt{<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 ULT 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 paper.

### 2.3. Algorithms and protocol details

The shared hash table \( H \) stores items that consist of a \(<\text{key}, \text{value}>\) pair \(<k, v>\). The hash table supports one operation only, defined as follows

\[
\text{access}(k, v) = \begin{cases} 
\text{if } <k, v'> \in H_{\text{pre}} \text{ then } & H_{\text{post}} = H_{\text{pre}} - <k, v'>; \text{ return}(v') \\
\text{otherwise} & H_{\text{post}} = H_{\text{pre}} + <k, v>; \text{ return}(\bot)
\end{cases}
\]

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

In a concurrent setting, we require the hash table to be linearizable [15]. Informally, this means that the operations are atomic and appear to be executed instantaneously at some point in wall-clock time between the start and end of the operation. Linearizability is composable, which allows us to correctly use the hash-table to implement other concurrent objects. In particular, this ensures that MPI calls take effect in program order.

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 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.

We describe below the eager protocol for blocking sends and receives then discuss the issues arising due to non-blocking calls and how to address them.

#### 2.3.1. Eager protocol

The pseudo-code for eager protocol is listed in Algorithm 1 on Page 10 and Algorithm 2 on Page 11 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. Only the receiving algorithm needs some elaboration.

A pair of an arriving message and a matching receive causes two accesses to the hash table: one by the communication server 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. An insert performed by a first-arriving communication server stores in the hash-table a pointer to the arrived packet. An insert performed by a first-arriving worker thread stores in the hash-table a pointer to a request. The ThreadWait operation is descheduling the invoking ULT; the ThreadSignal operation marks the ULT as runnable.

For each communication, our algorithms 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.
2.3.2. Rendezvous protocol

A rendezvous protocol usually involves the exchange of two control messages: a RTS (ready-to-send) message issued by the sender when the send is posted; a RTR (ready-to-receive) message returned by the receiver when the send and receive have been matched on the receiver side; after which data is sent directly from send buffer to receive buffer. The data transfer can be supported efficiently using the Remote Memory Access (RMA) feature of modern NICs [16].

Since we do not support wildcards, the matching of send and receive can occur on the sender side and the RTS control message can be avoided; the protocol is analogous to the eager-protocol, with roles reversed. The receiver sends the RTR message to the sender when the receive call is invoked by a worker thread; the communication server on the sender side accesses the local hash table with the RTR key. The sending worker accesses the local hash table with the RTS key. These two accesses can occur in either order, depending on the relative timing of the send and the receive calls. The first access inserts an entry in the hash table; the second results in a match and the RMA transfer is initiated by the UTL that made this second access. The communication servers on each side will mark the communicating ULTs as runnable when the RMA completes.

2.3.3. Non-blocking communication

Non-blocking communication in MPI is often used for overlapping communication and computation. The communication can be started but not finished using non-blocking calls such as such as MPI_ISEND or MPI_Irecv, which also create a request object that identifies the call. While the communication is in flight other computations can be performed to achieve the overlap. Finally, to complete the started communication, MPI_Wait for a single request, or MPI_Waitall for multiple requests (or other similar calls) can be performed. We refer to the first non-blocking call as comm begin, and to the second blocking call as comm end. MPI does not require that the comm begin and comm end calls be issued by the same thread; one thread can complete a non-blocking communication started by another thread.

We can split the execution of any blocking communication call into three phases, as follows:

- Phase (1) Initialization: The part of the execution which outcome does not depend on the state of the communication — i.e., does not depend on the timing of the matching call. E.g., setting up a request, or acquiring a packet for sending control data.
• Phase (2) *Posting*: The part of the execution which outcome depends on the state of the communication (i.e., the timing of the matching call) — but does not necessarily complete the communication. E.g., inserting the request into the hash table.

• Phase (3) *Completion*: The part of the execution that completes locally the communication, if it was not completed in phase (2). E.g., waking up a blocked ULT.

Consider now a non-blocking communication. Clearly, Phase (1) and (3) will be executed with the comm_begin and comm_end invocations, respectively. Phase (2) could be executed with either. For example, if the eager protocol is used, we can send data as soon as an MPI_ISEND is invoked or we can delay the transfer until MPI_WAIT is invoked. We call the first alternative *eager progress* and the second one *lazy progress*.

There are two advantages to lazy progress: First, it simplifies the case where a ULT which invokes comm_begin is not the one that invokes comm_end: When comm_end is invoked, it is known which ULT is waiting for the communication to complete; namely, the ULT that made the invocation. This information can be inserted in the hash table, so that the communication server will be able to signal the blocked ULT when a match is found in the hash table. A more complex protocol that leads to more cache misses is required, otherwise. Second, it enables a more efficient implementation of MPI_WAITALL: The passed arguments identify which communications will be completed by the call, and their number. Relevant entries in the hash table can be associated with a counter, and the communication server only needs to decrement this counter when a match is found.

Lazy progress has been the design of choice for early MPI implementations for the simple reason that the “progress engine” (in our case, the communication thread) had to be multiplexed on the same physical thread as the computation engine; short of using interrupts, it was simpler to delay communication thread operations to the point where a blocking MPI call is executed.

An advantage of eager progress is that it provides better overlap of communication and computation and enables earlier matching of sends and receives. This is most important for long messages that use the rendezvous protocol.

Our empirical study shows that better performance is achieved by associating phase (2) with the comm_end call, when the eager protocol is used. The balance changes for long messages that use the rendezvous protocol: Phase (2) enables the RMA data transfer and it is important for it to start as soon as the *start* calls of the matching send and receive have occurred. Hence, phase (2) should be executed with the comm_begin call in this case.

**Algorithms 3** on Page 15 and **Algorithms 4** on Page 16 describe our methods for comm_begin and comm_end for the eager protocol, when Phase (2) is performed with comm_end and comm_end is a call to MPI_WAITALL. The handling of MPI_WAITANY is similar.

The comm_start call merely creates a request object. The comm_end call takes a list of requests, and the number of elements in that list. The call completes all sends in the list and any receive that is already posted in the hash table; it associates the remaining receives with a counter that contains their number. If this number is positive, the invoking ULT waits for that counter to reach zero.

When the communication server receives a message that matches one of the pending receives, it decrements the counter and wakes up the blocked ULT if the counter is equal to zero. This signal operation at the communication server requires an atomic read and decrement operation on the counter. However, this is a local operation for most of the time, and the cost is minimal.

### 3. Runtime implementation and optimization

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.

#### 3.1. Concurrent hash-table

We use a chained hash table with linked lists. The default hash function is the FNVHash [17] which has been shown to be quite robust and fast compared to other hash functions for 64-bit value [18]. 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. **Fig. 2** illustrates an example of our hash-table.
Algorithm 3 Comm_begin and comm_end operations for a worker ULT

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm_begin(op, b, s, k)</td>
<td>( r = \text{op}.\text{ReqInit}(b, s, k); )( \triangleright ) Phase (1), request ( r ) is marked pending</td>
</tr>
<tr>
<td>return ( r )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm_end(requests[], count)</td>
<td>( \triangleright ) array of request, number of requests</td>
</tr>
<tr>
<td>so.thread = current ULT id</td>
<td>( \triangleright ) so is Sync object with a counter</td>
</tr>
<tr>
<td>so.count = count</td>
<td></td>
</tr>
<tr>
<td>for ( r ) in requests do</td>
<td>( \triangleright ) Phase (2), check requests for pending/finish</td>
</tr>
<tr>
<td>if ( r.\text{op} = \text{send} ) then</td>
<td></td>
</tr>
<tr>
<td>SEND-EAGER(r,b,r,s,r,k)</td>
<td></td>
</tr>
<tr>
<td>AtomicDecrement(so.count, 1)</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>( \triangleright ) receive</td>
</tr>
<tr>
<td>( v = (r.b, r.s, &amp;so) )</td>
<td></td>
</tr>
<tr>
<td>( v' = H.\text{access}(v, r.k) )</td>
<td></td>
</tr>
<tr>
<td>if ( v' \neq \bot ) then</td>
<td>( \triangleright ) matching send has arrived</td>
</tr>
<tr>
<td>Copy ( v' \rightarrow p.b ) to ( b )</td>
<td></td>
</tr>
<tr>
<td>pkpool.free(( v' \rightarrow p ))</td>
<td></td>
</tr>
<tr>
<td>AtomicDecrement(so.count, 1)</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>if so.count &gt; 0 then</td>
<td>( \triangleright ) Phase (3), complete or wait for completion</td>
</tr>
<tr>
<td>ThreadWait(so)</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>return Done</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 4 Lazy progress packet handler for communication server

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recv-EAGER-Lazy(p)</td>
<td>( \triangleright ) store pointer to packet data</td>
</tr>
<tr>
<td>( v' = H.\text{access}(p.k, p.v) )</td>
<td>( \triangleright ) match found</td>
</tr>
<tr>
<td>if ( v' \neq \bot ) then</td>
<td></td>
</tr>
<tr>
<td>Copy ( p.b ) to ( v' \rightarrow b )</td>
<td></td>
</tr>
<tr>
<td>pkpool.free(p)</td>
<td></td>
</tr>
<tr>
<td>if SubAndFetch(( v' \rightarrow so.count, 1 )) = 0 then</td>
<td></td>
</tr>
<tr>
<td>ThreadSignal(( v' \rightarrow so.thread ))</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
<tr>
<td>end procedure</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2. 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.
With the above optimization, the access operation has close to one cache miss, on average.

3.2. 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 [19]. 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 $\omega$. When a ULT starts running on the worker, it is also assigned a unique id $\Gamma$. A pair ($\omega$, $\Gamma$) uniquely defines a ULT in the system at a point in time. Since ULTs do not migrate, we can maintain a separate bit-vector for each worker; $\Gamma$ is the index in the bit-vector structure for the bit indicating the status of the corresponding ULT (runnable vs. running_blocked).

Algorithm 5 further describes the bit-vector scheduler.

```
Algorithm 5 Thread scheduler using bit-vectors
1: procedure SCHEDULING($\omega$, V) $\triangleright$ worker, bit-vectors
2: while !V.stop do $\triangleright$ loop until user ask to stop
3:     for word in V do
4:         if word $\neq$ 0 then
5:             localWord = 0
6:             AtomicExch(word, localWord)
7:         while localWord $\neq$ 0 do
8:             b = LeadingBit(localWord)
9:             localWord = FlipBit(localWord, b)
10:            ContextSwitch(b)
11:        end while
12:     end if
13: end for
14: end while
15: end procedure
```

In contrast to the 1-thread granularity of the queue structure, Fult scheduler works at 64-threads granularity. By using an atomic exchange instruction to swap the relevant word to a local variable, we are able to continuously perform read/write from/to this variable without accessing the main memory. An improvement in the instruction set (for example read-modify-write at the bit-level) could further improve this implementation.

A ULT executing a ThreadWait will invoke the scheduling code. The scheduler then looks for a new runnable ULT. ThreadSignal is a single atomic bit-set instruction (e.g., lock bts in x86). To facilitate more general functionality, we also implement a ThreadYield, which is simply a ThreadWait followed by a ThreadSignal on self.

The implementation requires the worker to iterate over the bit-vectors word by word to find a runnable ULT. This is efficient for up to 512 threads per worker, since $8 \times 64$-bit words still fit in a cache line. To support even more threads, we use a hierarchical bit-vector structure. More specifically, we use first level bit-vectors as “hint” to index into the second level bit-vectors. That is, each bit in the top-level bit-vectors indicates which bit-vector at the next level may have a bit on. A ThreadSignal performs a bit-set first into the lowest level then the higher one. On the other hand, the scheduler finds a thread by first looking at the top level then down to a lower level. The idea is similar to Bloom Filter technique [20], a query to the higher bit-vectors returns “possibly set” or “definitely not set” (or being set) for a lower bit-vectors.

If bit-vectors have $s_1$ bits at the top level and $s_2$ bits at the lower level, then the total number of threads we can support is $s_1 \times s_2$. We can choose $s_1 = s_2 = 512$ to fit each bit-vectors group in a cache line. Hence, a maximum of 256K concurrent ULTs per worker is supported.

Our scheduler does not schedule ULTs in strict FIFO order but provides a weak fairness guarantee: There is a constant $c$, so that if a ULT is enabled, then at most other $c$ ULTs will execute before it is scheduled. To show that this is true for a single-level table, consider the atomic exchange as taking a snapshot of the global state. In this snapshot, if a ULT is marked, it will be scheduled after all ULTs having smaller index are scheduled, which is bounded by the total length of the bit-vector. A slightly more complex argument works for a two-level table.
3.3. Concurrent packet pool

In general, the packet pool 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. In a single-threaded environment this design is sufficient for good performance, but not with multiple cores. 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.

Fig. 3 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.

Analyzing the packet life cycle naturally motivates a new design for the packet pool: split the centralized packet management into a private pool per worker. Initially, there is a fixed number of packets for each worker. At runtime, we allow moving packets among those pools via resource-stealing, similar to a non-blocking work-stealing algorithm [21]. Hence, we refer to our implementation of the pool as steal.

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. We use popTop for sending packets, and popBottom for receiving packets (used by the NIC) and for packet stealing (used by other workers). In case its private pool is empty, a thread steals packets from a randomly chosen private pool. Currently, we implement the pool using a simple ring-buffer and a spinlock.

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 ULT yields), which also places a limit on the number of pending requests for congestion and flow control [22].

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/creating a value in the container; and in the rare case of resource stealing, there could be more due to cache misses between processors.
4. Performance evaluation

4.1. Experimental setup

All of our experiments are done on the Stampede cluster and Stampede Knights Landing cluster [23] 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 16 GB 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.

4.2. Component overheads

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.

4.2.1. 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 Fig. 4.

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.

4.2.2. Concurrent hash-table

We measure the latency of hash-table operations in the two following scenarios, performing them with multiple POSIX threads:
Each improvement also shows invalidation. Experiments find higher variance at worse times, 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 versions of the hash-table (concurrent hash-table typically provides insert, erase, update, find, operations), we emulate that using a combinations 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.

Figs. 5 and 6 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.

• 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.

(a) Latency per successful access (there is no existing entry).
(b) Latency per failed access (there is an existing entry).

Fig. 5. 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 $2^n$ 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).

Fig. 6. 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 $2^n$ so that no extension is required during the experiment. TBB is also compiled with tbb-malloc to improve performance.
The overheads from wide lock-free threads. The evaluate 4.2.4. programming a on ulded 4.2.3. regime 

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.

4.2.3. Thread scheduler

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 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.

4.2.4. 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 Fig. 7, in comparison with common implementations using a concurrent lock-free stack and a lock-free queue available in the Boost library [26] (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 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.

4.3. Micro-benchmarks

In the previous section, we have analyzed our critical path by evaluating each individual component independently. The overheads are summarized in Table 2. Further, our performance results suggest that these overheads should not increase.

Table 1

<table>
<thead>
<tr>
<th>Processor</th>
<th>POSIX threads</th>
<th>Argobots</th>
<th>Fult</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SB</td>
<td>KNL</td>
<td>SB</td>
</tr>
<tr>
<td>Scheduling</td>
<td>0.75</td>
<td>4.5</td>
<td>0.05</td>
</tr>
<tr>
<td>Signal</td>
<td>1.15</td>
<td>4.5</td>
<td>0.30</td>
</tr>
<tr>
<td>Total</td>
<td>1.90</td>
<td>9.0</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Fig. 7. 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.
Table 2
Summary of the lower-bound performance number in usec for each component of our MPI communication layer.

<table>
<thead>
<tr>
<th>Component</th>
<th>SB + Mellanox</th>
<th>KNL + Omni-Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication (64 bytes)</td>
<td>1.0</td>
<td>2.8</td>
</tr>
<tr>
<td>Thread scheduling (fult)</td>
<td>0.03</td>
<td>0.13</td>
</tr>
<tr>
<td>Packet Allocation (steal)</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Message Matching (hashtable)</td>
<td>0.05</td>
<td>0.3</td>
</tr>
<tr>
<td>Total</td>
<td>1.18</td>
<td>3.43</td>
</tr>
</tbody>
</table>

Table 3
Summary of MPI configurations used in the evaluation.

<table>
<thead>
<tr>
<th>_Msg matching</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>m vapich2, intel</td>
<td>Queues Single</td>
</tr>
<tr>
<td>(m vapich2, intel)+async</td>
<td>Queues Single + Progress</td>
</tr>
<tr>
<td>(m vapich2, intel)+mt</td>
<td>Queues POSIX threads</td>
</tr>
<tr>
<td>pthread+hash</td>
<td>Hash-table POSIX threads</td>
</tr>
<tr>
<td>abt+hash</td>
<td>Hash-table Argobots</td>
</tr>
<tr>
<td>fult+hash</td>
<td>Hash-table Fult</td>
</tr>
</tbody>
</table>

![Fig. 8. Latency comparison for single-threaded OSU benchmarks.](image)

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 3 shows the different configurations that we evaluate. For MVAPICH2 and Intel MPI, we evaluate single threaded mode (m vapich2, intel); single threaded mode with asynchronous progress thread, i.e., MPICH_ASYNC_PROGRESS=1 (m vapich2, intel)+async); and multi-threaded mode ((m vapich2, intel)+mt). Using an asynchronous progress thread ((m vapich2, intel)+async) usually reduces performance quite a lot, as shown in Fig. 8; 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).

4.3.1. OSU latency benchmarks

We use the OSU benchmarks [27] 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 [28]. Further, in multi-threaded tests, we modify the code so that each ULT uses different tags.

RDMA data is ensured to be delivered in order (last byte comes last). This only affects the result of single-threaded benchmarks.
OSU single-threaded latency. Performance for the OSU single-threaded test is shown in Fig. 8. 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.

Our performance in both clusters, for messages of 64 bytes, is within 0.1 usec from the totals in Table 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 Fig. 9. 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 Fig. 5b 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. That still holds true even with 16 K 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.

4.3.2. 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_ISEND, 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 3, except “async”.

![Latency comparison between different MPI implementations using OSU multi-threaded latency test on the S8 cluster.](image)
The results for both SB and KNL cluster are shown in Fig. 10. The performances of the different implementations do not differ much in the SB cluster: full+hash achieves 3.0 Mmsg/s at 64-bytes, 20% better than pthread+hash, 3% better abt+hash and 5% better than mvsapich and mvsapich+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.

4.3.3. 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 106 messages equally divided among ULTs.

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. Fig. 11 presents our results for both SB and KNL cluster using different MPI implementations for this benchmark. In the SB cluster, our MPI using full+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 full+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, full+hash is 10% (8%) better than abt+hash, 7 \times (2 \times ) better than pthread+hash and 3000 \times (15 \times ) 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.

**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 **Fig. 12**. 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 **Fig. 12**, 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 assumes 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.

---

**Fig. 11.** Multi-threaded message rate benchmark performed in SB and KNL cluster.

**Fig. 12.** 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 **Fig. 11** - 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 total time (second y-axis) for each cluster.
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_mebw_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 Fig. 13. 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 full+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 MVAPICH2 implementations again suffer significant drop in performance when the number of threads increases; compared to the result using blocking calls in Fig. 11, 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 compute 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.

4.4. Mini-applications.

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.

4.4.1. NAS Parallel benchmarks - data traffic (DT).

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 number 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 [29].

The results are shown in Fig. 14. When there is an imbalance in the number of sources and sinks, we perform better in all cases, with up to $3 \times$ 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 pooled 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 full-l hash, respectively as reported by the perf profiler).

4.4.2. Breadth-first-search (BFS)

BFS is the kernel for the Graph500 benchmark [30], 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. [31] 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 downside, 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 TPS are provided in Fig. 15. We do not show the result for pthread-l hash since the performance is far worse ($10 \times$ 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 $3 \times$ performance over the reference code and 20% better than Argobots.

4.4.3. Unbalanced tree search (UTS)

Unbalanced Tree Search is a benchmark for evaluating the performance of parallel systems under heavily unbalanced and irregular workloads [32]. 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 [33–35]. 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 [36] 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 Fig. 16 for T3XXL tree (a 2.8 billion vertices, Binomial tree that is recom-
mended 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. Related work

Although MPI is still the de-facto programming model in High Performance and Scientific Computing, there is a great deal of research in new programming models as we start to think about exascale. A common theme of these new models is the ability to support intra-node parallelism using multi-tasking or fork-join models [37], coupled with some form of distributed memory communication. For example, PPL—a recent programming model developed at University of Illinois—provides a message-driven, multi-threading runtime in distributed memory using one-sided communication and software cache [38]. A survey of modern programming models and features is also provided in the paper [38]. These programming models are alternative options for MPI, and our techniques of implementations are applicable to them as well.

A common criticism of MPI is its inability to cope with increasing intra-node parallelism. Our work and the work of others show that the reason for these criticisms are less due to the message-passing model but rather to specific choices in current MPI definition and implementations.

There are several efforts to provide and improve multithreading support in MPI; we name a few here. MIMPI [39] and MPICH-MT [40] are early designs and implementations of thread-safe MPI on distributed memory. Another effort from Dózsa et al. [41] and Balaji et al. [6] studies the replacement of MPI coarse-grain lock by fine-grain locks and implement parallel receive queues using these locks; they demonstrate improvement in message rate up to 4 threads; however, performance suffers from mutex overheads. The change requires a complex, error-prone implementation (according to the authors). A more in-depth analysis of locking contention in MPI+Thread can be found in [8]. Recently, Intel researchers [9] also identified message matching as an important issue in MPI implementation and came up with a solution based on a hash-table. Their work was not focused on multi-threading, and they did not provide performance results for multi-threading. We shall be interested in comparing our work with theirs once their code becomes available.

A different approach followed by many projects is to implement MPI processes as threads [42–46]. This largely solves the performance issues we discussed, since each thread has a different rank and can have its own private MPI data structures. On the other hand, this increases the number of ranks, increases memory usage, and does not support dynamic thread creation; it is a different programming model.

Off-loading MPI communication or network polling to dedicated cores/threads is another theme of research. MPICH-Madeleine [47] 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 more costly than our design. The approach is more often used when MPI communication is integrated into a light-weight threading runtime such as Habanero-C [33], or Qtthreads [48]. Liu et al. [49] 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. We believe that once MPI is aware of concurrent executions and designed with that in mind, many of these techniques will not be necessary. For example, in our design, there is no need for an efficient concurrent queue (which is more conflict-prone than a hash-table) for passing requests to the server (which is required by [33]), and there is no need for calling MPI_PROBE or MPI_TEST as [33,48,49].

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 [50], MT-MPI [51], and pioman [52] 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 [22]. Our packet pool design has already taken that into account.

6. Discussion

The results in this paper raise two main questions:

1. Are the restrictions we impose on MPI reasonable; or do they prevent the use of important communication motifs?
2. Is it possible to relax the restrictions we imposed on MPI without losing the performance benefits of our approach?

We address each of these questions in turn.

6.1. MPI restrictions

Wildcard receives are often used, with MPI_ANY_SOURCE, in order to handle messages from different sources as soon as they arrive (thus coping with variable arrival times), or to support the one master, many slaves or one server, many clients motif in a scalable manner. On the other hand, the use of of MPI_ANY_TAG is very rare (we are not aware of any code using such).

We conjecture that the logic of application codes is such that it is seldom, if ever, the case that the same send operation is matched sometimes by a receive with a specific sender value and at other times by a receive with a wildcard sender
value; the sender “knows” whether the matching receive will use a wildcard. If so, we can segregate communications where wildcards are used from communications that do not use wildcards; e.g., by using different communicators. A message sent on a “wildcard” communicator is stored in the hash table with \( \text{key} = \langle \cdot, \text{tag} \rangle \), same as a matching wildcard receive. Thus, if our conjecture is valid, wildcards can be supported with no loss of expressiveness or performance and minimal inconvenience to programmers.

The constraint of having each send possibly match only one receive and each receive possibly match only one send is less important and can be relaxed without significant changes in our implementation: We will need to ensure that concurrent accesses to the same bucket in the hash table traverse every entry in the bucket atomically; an access either deletes the first matching entry or appends the new entry to the end of the bucket list. Performance will degrade when many concurrent communications use the same \( \langle \text{communicator}, \text{sender}, \text{tag} \rangle \) key, but we conjecture that such situations are rare and can (and should) be avoided. This proposed design does not guarantee that matches occur in the order required by MPI. Proper ordering, when needed, can be enforced by application programmers using suitable tag values; it can be enforced by MPI using sequence numbers.

We assumed that ULTs do not migrate once they started executing. ULT migration can be supported: It is essentially equivalent to deleting a bit from one bit vector (marking the ULT as not runnable on the source worker); and inserting a bit into another bit vector (as done when a new ULT is spawned). The overhead occurs only when ULTs are migrated—presumably an operation that is less frequent and has a relatively large overhead.

6.2. Extensions

A full implementation of MPI message passing has to support datatypes; has to support multiple point-to-point communication modes; and has to handle a variety of special cases, such as specified by MPI_STATUS_IGNORE or MPI_PROC_NULL. Handling these issues increases the latency of executing sends and receives, but does not require changes in the basic design outlined in the paper, since it does not require additional coordination between threads; it will not worsen the performance of MPI_THREAD_MULTIPLE, as compared to MPI_THREAD_SINGLE.

Our tight integration between the thread scheduler and the communication layer may be problematic, in practice, since the thread scheduler needs to interact with other subsystems, such as the runtime of the language that is used for multithreading. The two can be separated—we only require an efficient implementation of ThreadWait and ThreadSignal. Our work indicates the need for standardizing an interface that supports such functionality with a very low overhead.

Another important extension is to support multiple communication servers in order to support higher message rates. As long as wildcards are not supported, this does not seem to raise major issues: For example, the hash table can be split into distinct hash tables, each handling a distinct range of key values; each communication server would be associated with a separate hash table. This, too, will be the subject of our future work.

6.3. Benchmarks

It would be very desirable to evaluate our design with full-fledged applications or, to the least, with representative mini-applications. Unfortunately such effort suffers from a “chicken and egg problem”: Applications do not use MPI_THREAD_MULTIPLE, since implementations are inefficient; and implementations are not focused on MPI_THREAD_MULTIPLE since better implementations do not benefit and may harm current application codes. We believe that MPI_THREAD_FUNNELED is not a sufficient answer for nodes with hundreds of concurrent hardware threads and for asynchronous task programming models. One important way of breaking this vicious cycle is to demonstrate the benefits of good MPI_THREAD_MULTIPLE support with mini-apps and full applications. We hope to be able to do so in the coming years.

7. Conclusion

It has become a common wisdom that using MPI_THREAD_MULTIPLE is a bad practice. In this paper, we have laid the foundation for overturning this statement, thus greatly extending the programmability and usability of MPI toward future architectures. We showed that using a more restrictive model of MPI send-receive, we can design a constant-time overhead message matching and delivery under highly concurrent executions without sacrificing performance of sequential usage.

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 results show that it is feasible and desirable to delegate the handling of all the asynchronous communication events to one or few dedicated communication threads. Such a design reduces cache misses and results in a cleaner code than the coroutine-style code currently used by MPI libraries. This design also facilitates hardware-software tradeoffs, as most
changes will involve transfer of functions between NIC and communication server. The results also indicate the potential advantage of a hybrid architecture with latency nodes and throughput nodes: The communication server thread(s) would clearly benefit from running on a dedicated latency optimized core.

The model of a communication server that handles incoming communications and dispatches tasks is quite standard in hybrid data-flow models and similar communication-driven task models, such as Charm++ [53], PLASMA [54] or HCMPi [33]. But such systems typically implement their communication server atop MPI or a similar communication layer; functionality is duplicated, and performance is lost. We believe that the core communication mechanisms in our implementation can support directly and more efficiently models that require communication-driven task scheduling. In fact the results presented in Fig. 13 show that, with a large number of light-weight threads, non-blocking calls do not achieve better communication performance than blocking calls.

Our future work will focus on enhancing the functionality of our system. 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.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357 and by a subcontract from Sandia National Laboratories. It was performed while the second author was at the MCS Division of the Argonne National Laboratory. We thank Ron Brightwell, Pavan Balaji, Nikoli Dryden and Alex Brooks for their help with this work.

References


