Toward million communicating threads

Hoang-Vu Dang
Department of Computer Science
University of Illinois at Urbana-Champaign
hdang8@illinois.edu

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
snir@illinois.edu

William Gropp
Department of Computer Science
University of Illinois at Urbana-Champaign
wgropp@illinois.edu

ABSTRACT

We present the design, implementation and analysis of an efficient runtime which supports of million concurrently communicating threads using Message Passing Interface.

1. INTRODUCTION

Message Passing Interface (MPI) has been implemented by several vendors. On Blue Gene machine, MPICH [3] is the de-factor implementation, likewise on Infiniband machine, we have MVAPICH [5]. Cray has their own customization on top of MPICH called CrayMPI; Intel similarly implements IntelMPI. OpenMPI is another effort from Open-source collaboration of several industry and academic vendors [2]. Each vendor relies on their best knowledge of the underlying computing machine and network architecture to optimize their implementation. For examples, the implementation might reduce latency by accessing directly the low-level network-API or optimize critical path using specialized instructions.

Recently, because of the increasing in the number of processor cores within a computing board, there raises a need of running MPI efficiently in threaded environments. Although MPI specification requires all of its procedures to be thread-safe, MPI implementations are not originally designed to optimize for using multiple threads: OpenMPI latest version still does not provide a stable support of MPI_THREAD_MUTLIPLE [6], MPICH suports thread-safety via a coarse-grain locking which essentially serialized all MPI code. The research to replace coarse-grain locking is still undergoning [1], but the progress is slow because the existing software stack is only optimized for single-core performance. For example, there are significant number of global variables shared among various component of a MPICH. Global variables which implements stack, queue or hash-table can be replaced with concurrent data structure, others might be protected with finer-grains lock. However, locks and concurrent data structure has overhead could be added to the critical path and reduces performance of single-threaded applications.

On the other hand, communication in multiple threads is troublesome for many reasons. Data movement between processors could cache trashing or false sharing if taken care of. Processors can becomes under-ultilize because threads execute I/O operations can be preempted. Resource management have to consider locality and efficient concurrent accesses since they are shared among processors. Last but not least, kernel thread context-switching is costly, increasingly with the number of threads. Several studies have reported significant overhead for performing MPI concurrently in many threads [4, 1].

Because of the lack of efficient multi-threaded implementation, to our best knowledge, there is no production application that uses the thread-safety feature of MPI. What has been ended up is application running MPI in a "Thread Funnelled" mode i.e. only one thread is allowed to execute MPI procedure; other threads are used only for computation. This leads to a fairly restrictive programming model, in which communication and computation code appears in different phases or procedures. This approach often uses MPI non-blocking features to allow overlapping between computation and communication and hope the MPI runtime will be able to make communication progress behind the scene. In practice, to support better asynchronous communication, user of MPI implementations often need to enable a flag to request the runtime to spawn a dedicated polling thread for progressing the communication (e.g., MPICH_ASYNC_PROGRESS flag in MPICH/MVAPICH).

The solution that we have seen for existing MPI implementation to adopt multi-core machine is thus ad-hoc and inefficient. In this paper, we present an approximate implementation of MPI point-to-point communication which takes into account of multi-core machine by design. Our implementation is approximate since we shall relax some semantics requirement of MPI. We argue that our design and algorithms for this relaxation is able to achieve efficient implementation, yet do not restrict applications from using MPI effectively. More complex MPI semantics and functionality can be built on top of our point-to-point communication. By using this bottom-up strategy, we provide an insight on the minimal cost to implement MPI, accurately to the number of memory accesses. We summarize our most signfinicant contributions as follows:

- A light-weight thread scheduler using bit-vector which achieves single-instruction for signalling continuation.
- An constant time overhead algorithm for MPI point-to-point communication utilising a specially designed

Euro MPI Aug XX–YY, 2016, XXX, YYY, ZZZ

DOI: 10.475/123_4

concurrent wait-free hash-table.

- A resource-aware locality-aware concurrent memory pool for packet management.
- A MPI runtime design for scaling up to million communicating threads.

Our paper is organized as follows. The next section describes our runtime architecture from the high-level design to individual components. We present our implementation details and optimization in Section 3. Section 4 discusses experiment and results. Section 5 gives an overview of some related works. Finally Section 6 concludes our study.

2. RUNTIME ARCHITECTURE

2.1 High-Level System Design

We first identify the complication of MPI code is on the message matching mechanism. MPI point-to-point procedures (Send/Recv) facilitate matching messages by tagging each with an integer in addition to destination and communicator. The implementation uses several queues data structure to store incoming requests and messages. For example, the unexpected queue store arrived messages without a matching request; the posted queue will store the pending requests without matching messages. Depending on different arrival order, traversing, insertion or deletion of entries into queues will be issued. In a concurrent setting, these are critical section that need to be protected. An efficient lock-free control of these operations are non-existent even though individual queue could be implemented efficiently. Moreover, when the number of concurrent communication increases, traversing a queue requires linear time to the size of pending requests.

We propose an implementation MPI based on the following assumptions:

- Large number of concurrent threads Threads are lightweight; they are scheduled by a user-space scheduler that understands synchronization objects.
- The implementation is optimized for the case where MPI_ANY_SOURCE and MPI_ANY_TAG are not used.
- No concurrent send and receive having the same signature, thus we do not yet worry about odering.
- The 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 only x86_64 machine, however the technique is general enough to port to other architecture which supports atomic exchange and atomic bit manimupation.

The key data structure for communication is a lock-free hash table which stores both unsolicited incoming messages and outstanding receives. Since we assume there are no don't cares, we can hash by source and tag. We focus on critical operations, the creation of communicators or of datatypes is not (yet) address. We also ignore, for the time being, one-sided operations.

The runtime is an integrated system of a user-level thread (ULT) scheduler and a communication server. The ULT provides a low latency thread management while the server is a kernel-thread dedicating to message delivery. The two components interact using the afortmentioned hash-table. Our primary goal is to design an algorithm and optimize to minimize the latency of this interaction. A failure to achieve this goal simply adds extranous overhead to single threaded application and might not achieve significant benefit that justifies its cost. We shall also show that our design allows optimization space to be reduced to optimizing a few neccessary operations.

Resource management is another important issue. Aside from communication context such as connections and NIC-provided control data which we maintains per communication server, we have to maintain a pool of packets structure. The pool is the second shared data structure of the runtime. Each packet is used for storing control and message data to deliver to network device.

Figure 1 shows the overall architecture of our described runtime system.

2.2 Point-to-Point Message Passing Algorithms

We now describe an algorithm and its data structure that facilites the point-to-point communication in Message Passing semantics. Our algorithm relies on a specialized concurrent hash-table H defined as follows.

We denote a tuple $(k, v) \in H$ when v is stored in the hash-table using the key k. At initialization, for any key k, only the tuple $(k, \perp)$ are stored in the table. The hash-table has two operations:

- H.insert(k, v): attempt to replace value of key k by v.
- H.empty(k): replace value of key k by $\perp$

Additionally, let H_t denote a state of H which is a set of all key-value pair stored in H at time t, H_{t_0} denote the state of H before a operation and H_{t_1} denotes the state of H after a operation; $\mathbb{K}, \mathbb{V}$ denote the key and entry space. In a sequential history, we have the following legal semantics:

$$\text{insert}(k, v) = \begin{cases} v & \iff (k, \perp) \in H_{t_0}, (k, v) \in H_{t_1} \\ v' & \iff (k, v') \in H_{t_0}, (k, v') \in H_{t_1} \end{cases} \quad (1)$$

$$\text{empty}(k) = \text{success} \iff (k, v) \in H_{t_0}, (k, \perp) \in H_{t_1} \quad (2)$$

$$\forall k_0 \in \mathbb{K}, \nexists (v_0, v_1) \in \mathbb{V} \mid v_0 \neq v_1 \wedge (k_0, v_0), (k_0, v_1) \subset H_t \quad (3)$$

That is, (1) means that `insert` succeeds only if the entry being stored with k at the time of insertion is $\perp$. In that case, $(k, \perp)$ is replaced with (k, v). Otherwise, the operations fail and existing value is returned i.e. no changed are made to H. In contrast, `erase` as in (2) is always successful, which replaces any value v with the input k with $\perp$, essentially removing v from the hash-table. The (3) equation is the consistency requirement for hash-table, which means for each key we allow only one value associated with it.

In a concurrent setting, we further require the hash-table to be *linearizable*. This means we ensure firstly safety and correctness property. Secondly, we ensure operations take affect in a real-time order. This is a strong guarantee however is neccessary to implement MPI semantics which have

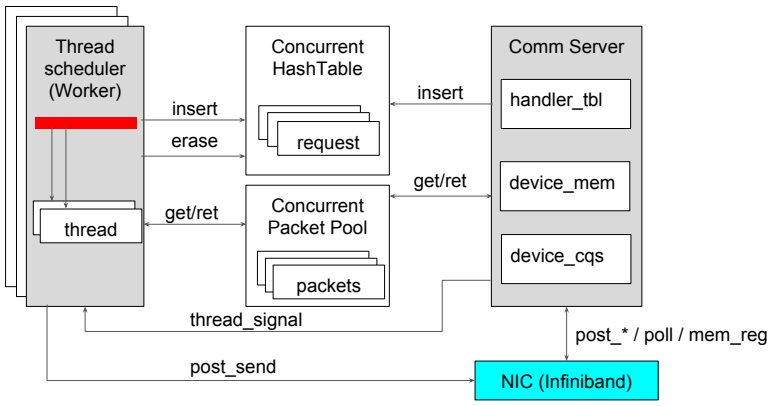

Figure 1: MPI Runtime Architecture for multi-threaded executions

operations executed in program order. Lastly, linearizability is *composable* which allows us to correctly use the hash-table to implement other concurrent objects.

Message delivery in general can be implemented in two way: *eager* or *rendevouz* protocol. Eager protocol is used when the target buffer is not known to the sender. Thus, an copy is made into an intermediate buffer - usually packed with other control data to deliver to the network. This also allows the Send operation to return immediately as the buffer can be reused. This protocol however becomes inefficient when message size gets larger, typically larger than the L2 or L3 cache size i.e. the cost of data movement is significant. When this is the case, we switch to rendevouz protocol in which the data is delivered directly from the source buffer to the target buffer by the NIC thus saving extra copies. The protocol however requires some control messages to exchange control data and signal completion.

2.2.1 Eager protocol

The pseudocode for eager protocol is listed in Algorithm 1 and 2 for worker thread and communication server respectively. The basic idea here is that the thread and the communication server could use the results from hash-table insertion to coordinate the matching of messages and request.

If the communication server succeeds with the hash insertion, the server knows the receiving request has not been posted and it returns immediately. A thread later comes, eventually fails the insertion but finds a packet with the needed data to work with. On the other hand, if the thread is the one who succeeds, the request is then inserted into the hash-table with the synchronization object. The thread is yielded by executing `ThreadWait`. The worker is now free to do other works. When the associated packet arrived, the server will fail the insertion and find the request with the attached synchronization object. It now can mark the thread as schedulable using `ThreadSignal` so when the worker is available it will pick up the works with the available data.

Since there is no need for linear searching, our matching algorithm achieves a constant time complexity as long as operation of thread scheduler, pool and hash-table is also constant time.

2.2.2 Rendevouz protocol

Algorithm 1 Eager-message send/recv for thread

```
1:  procedure SEND-EAGER(b, s, d)          ▷ : buffer, size, destination signature
2:      p = pkpool.get()
3:      Set packet header p.d to d
4:      Copy b to p.b
5:      Post p to network for send.
6:  end procedure
7:
8:  procedure RECV-EAGER(b, s, d) ▷ : buffer, size, source signature
9:      Create a request r = (b, s, d, (ω, Γ))
10:     Create hash-value v from r.
11:     Create hash-key k from d.
12:     v' = H.insert(k, v)
13:     if v' ≠ v then                     ▷ : insertion fail
14:         Copy v'.p.b to b  ▷ : message arrived, copy data.
15:         pkpool.ret(p)
16:     else                               ▷ : insertion success
17:         ThreadWait()      ▷ : message not arrived, wait.
18:     end if
19:     H.erase(k)
20: end procedure
```

Algorithm 2 Eager-message packet handler for communication server

```
1:  procedure RECV-EAGER-PACKET(p)
2:      Create hash-value v from p.
3:      Create hash-key k from p.d.
4:      v' = H.insert(k, v)
5:      if v' ≠ v then                     ▷ : insertion fail.
6:          Copy p.b to v'.r.b  ▷ : thread arrived, copy data.
7:          ThreadSignal(r.Γ)
8:          pkpool.ret(p)
9:      else                               ▷ : insertion success.
10:         return      ▷ : thread not arrive, nothing to do.
11:     end if
12: end procedure
```

A rendevouz protocol have the same algorithm as short protocol after we have exchanged the control message via eager-protocol. The control messages includes two messages: a RTS (ready-to-send) issued by the sender, a RTR (ready-to-receive) issued by the receiver. By then, the sender and the receiver has known the addresses of each other buffer and they can perform communication. The data transfer could be optimized further using the Remote Direct Memory Access (RDMA) feature of modern Network Interface Controller (NIC). Several researchs has focused on optimizing this operation.

As an effect of our assumption, in this protocol we can save one control message i.e. the RTS. The reason is we do not have wild-card, the sender and receiver knows exactly their target. We only require the receiver to send its buffer to the sender. The sender can follow up by issuing an RDMA. Analogous to the eager protocol, whenever the sender or receiver is required to wait for a matching message it will perform `ThreadWait`, and later when the message has arrived the communication server will perform a `ThreadSignal`.

Specifically, the sender waits when the RTR message has not arrived or when RDMA is pending. In both situation the server wake up the sender thread when RDMA has completed. The receiver after issuing the RTR will wait until the signal that RDMA has finished arrived and data is now available.

2.3 Critical Operations Discussion

Clearly, optimizing the hash-table, the packet pool and the thread scheduler operations are the key to achieve good performance for our protocols. One could achieve $O(1)$ amortized in complexity however, an efficient implementation requires optimizing for the constant factor. Ideally, we want these operations to be *wait-free* to ensure progress and minimizing the number of memory accesses and possible cache misses. In the next section, we describe the implementaion of each of the three components and show how we could optimize toward our goals.

3. RUNTIME IMPLEMENTATION AND OPTIMIZATION

3.1 Thread scheduler

As mentioned earlier, our scheduler implements User-Level Thread (UTL). The user specifies the number of workers which are implemented with POSIX thread and pinned to a specific core. The context-switching mechanism is similar to that of Argobots and Boost coroutines, we make use of `fcontext`. The idea is to maintains a separate stack for each thread. A few registers (including the program counter) is saved and restored from an allocated space in the stack. Morever, the context-switching is designed to not involve Operating System (OS) system call, thus by-passing kernel code. This overall reduces the latency of switching to a new context to less than a hundred cycles.

Rather than designing a general purpose ULT, we focus on the two most important operations for synchronization purpose: `ThreadWait` and `ThreadSignal`. A possible implementation is *condition variable*. This is the typical mechanism in POSIX thread, and other ULT library. Condition variable is a generic container which can be used as a building block for many other synchronization primitives. However, it is still expensive as in general it requires a mutex lock and some queue traversal when many threads are blocking. An alternative is to use a busy-waiting synchronization flag which has lower latency. However this is considered even worst for scaling since the processor spends useless time polling. Qthreads implements a slight modification notion of condition variable namely *full-empty bit*. However, internally each object is a waiting queue and rescheduling requires expensive traveral of this queue to find runnable threads and insert back to run queues.

Our thread scheduler targets a constant time overhead for both afortmentioned operation. Marking a thread as runnable or waiting is only a single x86 instruction. The idea behind our thread scheduler is a novel use of bit-vector. Rather than using a queue to implement a set of runnable thread, each bit in the bit-vector indicates a schedulable thread. When a worker is created, it is given a unique worker id, denotes as ω. When a thread is created by a worker, it is assigned an unique id Γ within the range of $[0, M-1]$, for M is the maximum concurrent threads for that worker. For example, using a vector of 8 64-bit words, we allow up to 512 concurrent threads per worker. A pair (ω, Γ) uniquely defines a thread in the system at a point in time. During thread's creation time, the associated Γ bit in its bitvector structure is marked so that when the worker is free it will attempt to schedule that thread to run. Algorithms 3 further describes our scheduling algorithms.

Algorithm 3 Thread scheduler

```
 1: procedure SCHEDULING(ω, V)        ▷ worker, bit-vector
 2:     while !ω.stop do              ▷ loop until user ask to stop
 3:         for word in V do
 4:             if word ≠ 0 then
 5:                 localWord = 0
 6:                 AtomicExchg(word, localWord)
 7:                 while localWord64 > 0 do
 8:                     b = FindFirstSet(localWord)
 9:                     localWord = FlipBit(localWord, b)
10:                     ContextSwitch(b)
11:                 end while
12:             end if
13:         end for
14:     end while
15: end procedure
```

Algorithm 4 Thread Operations

```
1: procedure THREADWAIT
2:     ContextSwitch(ω)
3: end procedure
4:
5: procedure THREADSIGNAL(sync)
6:     AtomicBitSet(sync.ω.V, sync.Γ)
7: end procedure
```

The scheduler works at 64-threads granularity instead of 1-thread traditionally. By using an atomic exchange instruction to swap the interested word to local variable, we are able to continously perform read/write from/to this variable without accessing the main memory. This is not only ensures progress property for all threads because we can

schedule them in order, it solves the problem of having no atomic read-modify-write instructions operating at bit level.

Algorithms 4 describes the two thread operations. `ThreadWait` simply switch back to the worker scheduler based on current worker id (i.e. ω) On the other hand, `ThreadSignal` shall access the bit-vector and atomically set the bit in the appropriate word based on the the same information stored in the synchronization object. ω and Γ are stored in thread local-storage of the worker, and is updated whenever a new context is switch to.

There are couple of issues worth discussing on the above design. Firstly, note that `ThreadWait` is executed in the same kernel thread as the thread scheduler associating with the running ULT while `ThreadSignal` can be executed anywhere. Thus, It is errornous to perform `ThreadSignal` multiple times on the same object since they will be treated as one or more signal depending on the time the worker looks at the bit-vector. It is also erronous to `ThreadWait` when the associted `ThreadSignal` has been performed. In practise for each MPI request we use an additional flag to indicate whether the signal has been executed before performing the `ThreadWait`. This is important for non-blocking communication in the case of `MPI_Waitall` since the communication can be completed before the thread attempts to wait.

The next issue is that the performance shall degrades when we increase the maximum number of threads since iterating over the bit-vector word by word is more expensive. We tackle this issue by using a hierachical design of bit-vector. That is, we could use an additional bit-vector as a hint to index into the original bit-vector. Specifically, each bit in the secondary bit-vector indicates which word in the first level may have a schedulable thread. More specifically, `ThreadSignal` will perform a bit set first into the first level then a bit set into the second level. The scheduler will first look into the second level to find a potential word and go directly to that word to look for schedulable threads. Using this scheme, suppose we use one word for the secondary level bit-vector, each bit represents a 64-bit word in the first level, then each worker can support up to 4096 threads. And since eight words can fit in a cache line and better to read together, we can use 1 bit to represent a set of eight 64-bit. Similarly by having eight words in the secondary level we can support 262144 concurrent threads per worker. With a small number of worker, we can go up to our goal of million thread without sacrifying much performance. Also note that this is the maximum number of concurrent threads; when a thread completes its current task, it can be re-used for launching another task.

There is an fairness issue with the thread scheduler i.e. thread with lower index typically is scheduled before thread with higher index. There are a few solutions to improve fairness for example occasionally traverse the bit-vector in different orders. We however currently ignore this problem and leave it for future works. Having said that, this issue is not severe since our algorithms still ensure progress property. If a thread is marked as schedulable, it's eventually will be scheduled in a bounded number of steps. To show that this is true, consider the atomic exchange as taking a snapshot of the global state. In this snapshot, if a thread is marked it will be scheduled after all threads having smaller index are scheduled, which is bounded by the maxmimum number of allowed concurrent threads M.

3.2 Concurrent Hash-Table

Our concurrent hash-table is not deviated very much from conventional concurrent hash-table. We however find opportunity to optimize further by taking advantage of some semantics requirement that we have mentioned.

Firstly, we use a spinlock per bucket. This is an viable option since we could control the table size to reduce the hash collision to minimal. The table size is related directly to how many concurrent operations which is controlled by our packet pool size. When there is no collision, the conflict can only happen between communication server and a thread when both try to insert into the same bucket at the same time. With a small number of conflicts, a spinlock is sufficient for synchronization and allow a very simple design.

Secondly, we design each bucket as an 4-entry array. Each entry consists of two 64-bit words. Within each 4-entry, one entry will be used as *control entry* and the other three are used for *data entry*. The control entry has an atomic flag for spin locking, and a pointer to point to the next 4-entry bucket in case we have more than 3 collisions. A data entry consists of two 64-bit words of key and value pair. In total, 4-entry takes 64-bytes and typically fits in a cache line. Thus we suffer only 1 cache miss when trying to lock the bucket and the data can be read without more cache misses.

Lastly, the `insert` operation also returns the address of the associated entry having the key in memory. This allows the `empty` operation to be a single instruction/single memory write to set the value associated with key to $\perp$. This is only possible when only one thread can write to an entry key which is true when no concurrent communication with the same tag is allowed.

In conclusion, with the above optimization, the `empty` operation is lock-free and wait-free, costs only 1 memory write. The `insert` operation is cache-friendly and is typically wait-free, costs also 1 memory write in the common case of no hash collision (write is made to spin-lock; other accesses are in cache).

3.3 Concurrent Packet Pool

In general, the packet pool can be implemented using a lockfree stack. A pool `free` is translated to a stack `push`, and `alloc` is translated to a stack `pop`. At the initialization, a fixed number of packet is initialized from the main memory and `push` to the container. 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 for the case of multi-core/multi-threaded. Consider a packet recently being used by a thread and returned to the pool, this packet could be subsequently obtained by a different thread running in a different core. This leads to several cache misses since the cacheline is momently owned by the first thread. An example is when two threads running in two different cores alternatively perform `MPI_Send`.

Figure 2 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 can only has the affinity of either the communication server or one of the worker thread. More specifically, if the packet is used for sending data, it shall have the locality of the thread; if the packet is used for receiving data, it shall have the locality of either the thread

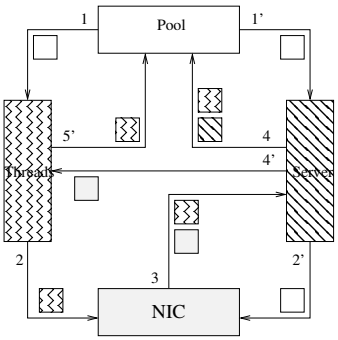

Figure 2: Packet life cycle: (1) A thread sending data obtain a packet from the pool; (2) The thread writes data into the packet and submit it to the NIC; (1', 2') Communication server obtains a packet for receiving data and posts the packet to the NIC; (3) The server polls the NIC and obtains the packet back (either send/recv packet). (4) If the packet is for sending, it is returned to the pool; If the packet is for receiving and there is a request, the server copies the data and returns the packet to the pool; (4') If the packet is for receiving and the there is no request, the packet is inserted to hash-table. (5') The thread takes packet from the hash-table copies the data before returning it back to the pool.

or the server depending on whether the associated request and the packet is received first. Moreover, when the packet is posted by the server to the NIC for receiving data, the packet shall lose any affinity it has since the NIC will perform a write over the packet.

Analysing the packet life cycle motivates us a new design for the packet pool. We split the centralized packet management into a private pool per worker and a shared pool per communication worker. Initially, there is a fixed number of packets for each of the pool. At runtime, we design an algorithms to allow moving packets among those pool depending on their usages. Our goal is to maintain good locality per hardware thread but does not block if there is available resources.

The private pool consists of packets that are identified as having affinity of the associated hardware thread. It 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 the top of the queue, and also allows poping from its bottom. The idea is that at the bottom of the deque are packets that are least recently used and are better candidates for other purposes. That is, when the private pool is full, it performs `popBottom` and pushed back to the shared pool.

When a packet for sending is needed by a thread, a `popTop` is performed with the private pool of the current worker, which guarantees giving the last packet that was read or written by the same worker. If the private pool is empty, the shared pool is accessed for extra packets. Alternatively, when a packet is needed for receiving data by the communication server, the server first try to obtains packets from its shared pool. If the shared pool is empty, the server now performs a *steal* by randomly chosing a private pool and perform the `popBottom` operation.

The private pools can be thought as caches, and the shared pool can be thought as the main memory. When "the cache" is full, it evicts the least recently used to the "main memory". Our heuristic essentially performs resource-balancing among the set of packets by moving it around pools with the locality taken into account. Since the private pool is only shared between a worker thread and the communication server, currently we implemented it using a simple fixed-size buffer with a spinlock. The shared pool is accessed by more than two threads, thus implemented using a lock-free stack.

4. EXPERIMENT

4.1 Thread Scheduler

We describe here an experimence to evaluate the overhead due to thread scheduling.

4.2 Concurrent Hash-Table

We describe here an experiment to evaluate the overhead due to accessing hash-table.

4.3 Concurrent Packet Pool

We describe here an experiment to evaluate the overhead due to packet pool.

4.4 Overall overhead

Overally this is the overhead.

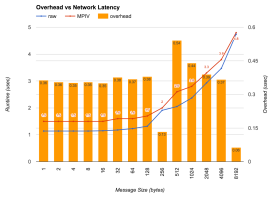

Figure 3: Total overhead of runtime system, comparing to raw network latency.

Overally this is the latency compare to OSU.

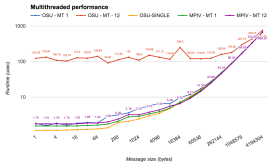

Figure 4: Comparing multi-threaded pingpong with OSU benchmarks using 1-thread in 1-worker and 12-threads in 12-workers. MPIV is our implementation.

Overally this is the message rate compare to OSU.

4.5 Applications

We shows BFS, UTS, Stencil.

5. RELATED WORKS

6. CONCLUSION

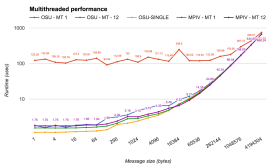

Figure 5: Comparing multi-threaded pingpong with OSU benchmarks using 1-thread in 1-worker and 12-threads in 12-workers. MPIV is our implementation.

7. REFERENCES

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward efficient support for multithreaded mpi communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129. Springer, 2008.

[2] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[4] W. Gropp and R. Thakur. Issues in developing a thread-safe mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 12–21. Springer, 2006.

[5] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design of high performance mvapich2: Mpi2 over infiniband. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48. IEEE, 2006.

[6] T. O. M. Project. Is open mpi thread safe. https://www.open-mpi.org/faq/?category=supported-systems, 2016. [Online; accessed 9-April-2016].