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 is the de-factor implementation, likewise on Infiniband machine, we have MVAPICH. 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. Each of the implementation uses their best knowledge of the underlying computing machine and network architecture to optimize their code. For examples, the implementation might reduce latency by accessing directly the low-level network-API or optimize critical path using specialized instructions to reduce cache misses.

Recently, because of the increasing in the number of processor cores within a computing board, there raises a need of running MPI efficiently in a threaded environment. Although MPI specification requires all of its procedures to be thread-safe, MPI implementations are not originally designed to optimize for using multiple threads. For example, OpenMPI latest version still does not provide a stable support of MPI_THREAD_MUTLIPLE. MPICH suports thread-safety via a coarse-grain locking which essentially serialized all MPI code. The research to remove these coarse-grain locks is still undergoning, but the progress is slow because the existing software stack is only optimized for single-core performance. For an 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.

Communication in multiple threads could also cause cache trashing or false sharing if not carefully be taken care of. Moreover, the processor can becomes under-ultilize because threads execute I/O operations can be preempted. Last but not least, kernel thread context-switching is costly. Several paper has report significant overhead for performing MPI concurrently in different thread.

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 applications runs 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 functions. This approach often uses MPI non-blocking feature to allow some overlapping between phases and hope the MPI runtime will be able to execute communication code 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. By using this bottom-up strategy, we provide an insight on on the cost to support a complete MPI, accurately to the number of memory accesses.

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 Network Interface Controller (NIC) supports multiple, independent channels.

- 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.

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 store (k, v) into the hash-table.

- H.empty(k): replace any value stored using key k with $\perp$

Additionally, let H_t denote a state of H in real 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:

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

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

$$\forall k_0 \in \mathbb{K}, \nexists v_1, v_2 \in \mathbb{V} \mid v_1 \neq v_2 \wedge (k_0, v_1) \in H_t \wedge (k_0, v_2) \in H_t$$

That is, the insert is only successful 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 constrast, the erase is always success, which shall replace any value stored with the input k with $\perp$, essentially removing the entry from table. The last point is the consistency requirement, which means for each key, we can find only one value associated with that key.

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 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 Point-to-Point Eager protocol

The algorithm for eager protocol is listed in Algorithm 1 and 2 for worker thread side and communication server side respectively. The basic idea here is that the thread and the communication server at the receiving side can effectively coordinating to match messages and requests.

If the communication server succeeds with the hash insertion, the receiving request has not been posted by a thread,
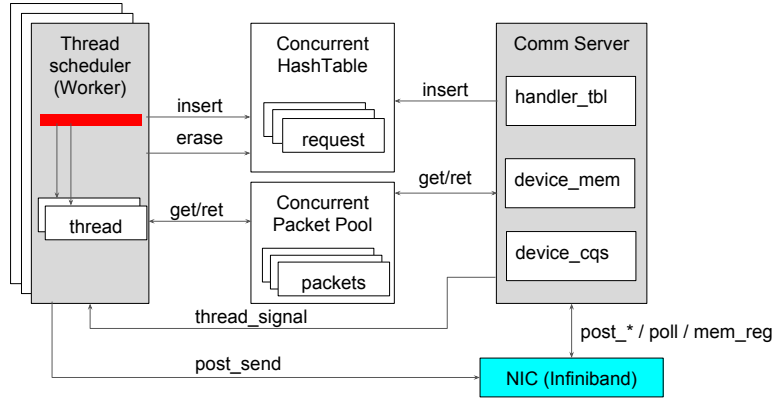
Figure 1: MPI Runtime Architecture for multi-threaded executions

---

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 $r = (b, s, d, (\omega, \Gamma))$
10:    Create hash-value v from r.
11:    Create hash-key k from d.
12:    $v' = H.$insert(k, v)
13:    **if** $v' \neq v$ **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** $v' \neq v$ **then**   ▷ : insertion fail.
6:        Copy $p.b$ to $v'.r.b$ ▷ : thread arrived, copy data.
7:        ThreadSignal$(r.\Gamma)$
8:        pkpool.ret(p)
9:    **else**   ▷ : insertion success.
10:        **return**   ▷ : thread not arrive, nothing to do.
11:    **end if**
12: **end procedure**

---

the server return immediately. A thread later comes, eventually fail the insertion but find a packet with the needed data to copy to its buffer.

On the other hand, the thread is the one who succeeds and the request is inserted into the hash-table with the synchronization object. The thread yield by executing `ThreadWait`. When the eager packet arrived, the request handler is executed by the communication server. The server will fail the insert but find the request with the attached synchronization object. It now can wake up the thread using `ThreadSignal`. In this situation, there is a locality consideration whether the thread or the server should perform the memory copy. However, this is an optimization which does not effect the correctness of our algorithm.

2.2.2 Point-to-Point Rendevouz protocol

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.

In our runtime, 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

Our thread scheduler is implemented as a User-Level Thread (ULT) in order to minimize context switching overhead. 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: `ThreadWait` and `ThreadSignal`. One possible implementation is condition variable. This is the typical synchronization often used in POSIX thread, and other ULT library such as Argobots. However, these are expensive since condition variable in general requires a lock. A busy-waiting implementation is even worst since the processor spends useless time polling. Qthreads uses a slight modification notion i.e. full-empty bit. However, internally each full-empty bit object is a waiting queue and rescheduling requires traversing this queue to place them back to the run queue.

Our thread scheduler achieves wait-freedom for both afortmentioned operation. Except the required context-swiching for `ThreadWait`, each operation is simply 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 thread is runnable.

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 the creation time, the worker atomically set the associated Γ bit in its bitvector structure.

Algorithms 3 describes our scheduling algorithms. The worker, when available will attempt to find a schedulable thread in its bitvector. Instead of working at 1-thread granularity, we work at 64-thread represented by a 64-bit word. By using an atomic exchange 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.

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
```

This is not only ensures progress property for all threads because we can schedule them in order, it is also provide correctness since there is no atomic read-modify-write x86 instructions operating at bit level. We do not support thread-stealing yet, but it can be done in similar fashion by stealing at 64-thread granularity.

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.

Note that `ThreadWait` is executed in the same kernel thread as the thread scheduler associating with the running ULT, thus there is no race that we need to worry about in accessing the bit-vector. `ThreadSignal` can be executed anywhere, and in our design it will be executed by the communication server. Thus in the algorithm it is errornous to perform `ThreadSignal` multiple times on the same object. In practise, we store additionally an atomic flag per synchronization object to prevent this to happen.

There are two possible problems with the described thread scheduler design. Firstly, the performance degrades greatly when we support a large number of threads. The reason is we have to iterate over the bit-vector. Secondly, there could be an issue with fairness because thread with larger bit-index will be scheduled later.

We currently ignore the fairness issue, and as long as we maintain the progress property this is not a big problem. We tackle the first problem by using a hierachical design of bit-vector. That is, we could use a secondary bit-vector as a hint to index into the first level bit-vector. Specifically, each bit in the secondary bit-vector indicates which word in the first level could have 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 potentially word and go directly to that word index to look for schedulable threads. It can happen that the secondary might indicate the thread is schedulable, but when it comes to the first level the thread is already scheduled by the previous check. This *false positive* happen because the secondary level bit is potentially shared a group of thread. However, it does not affect the correctness since the thread will not be scheduled twice in any situation.

Using this scheme, if we use 1 word for the secondary level bit-vector, each bit represent a 64-bit words in the first level, 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 words. Similarly by having eight words in the secondary level we can support 262144 concurrent threads per worker. Therefore, with a small number of worker, we can go up to our goal of million thread without sacrificing much performance.

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. Since the collision is minimal, the conflict can only happen between communication server and a thread when both try to insert into the same bucket at the same time. Since there is only synchronization between 2 threads a spinlock is sufficient for our purpose moreover allowing a very simple design.

Secondly, we design each bucket as an 4-entry array. Each entry consists of 2 64-bit words. Within a 4-entry array, one will be used as control entry and the other three are used for storing a pair of key and data. The control entry has an atomic flag for spin locking, and a pointer to point to the next bucket in case we have more than 3 collisions. Since a bucket typically fits in a cache line, we suffer only 1 cache miss at most when trying to lock the bucket and the data can be read without more cache misses.

Lastly, the `insert` operation returns the address of the associated entry. This allows the `empty` operation to be a single instruction, which set the key to ⊥. This is only possible because only one thread can change an entry key because no concurrent Send/Recv with the same tag is allowed.

Therefore our `empty` is lockless and achieves wait-freedom too. Our `insert` is cache-friendly and typically also wait-freedom with high probability when the table size is large enough (thus with little to no hash collision).

3.3 Concurrent Packet Pool

In general, the packet pool could be implemented using a lockfree stack. A pool `ret` is translated to a stack `pop`, and `get` is translated to a stack `pop`. The Last-In-First-Out (LIFO) property allows good temporal locality for copying data to and from the packet.

One problem with the described design is that a packet used by a hardware thread could be subsequently used by a different one because they share the same pool. This leads to several cache misses because the cacheline has become dirty by other threads modification. An example is when two threads from two different workers alternatively perform MPI_Send. The packet is modified by one worker and later obtained and modified by another worker after it has been returned to the pool. Additionally, when a packet is posted to the network as a recving buffer, it will be written by the NIC, thus losing its locality to any of the thread previously has. These observation leads to a better design of our packet pool: we split the packet management into two parts: a private pool per worker and a shared pool. Initially, there is a fixed number of packets for each private and shared pool. At runtime, we design an algorithms to allow moving packets among those pool depending on their usages.

The private pool consists of packets that are recently read or written by the worker (or 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 posting to the NIC receiving queues as buffers by the communication server, the server obtains packets from the shared pool first. 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. Since the conflict is only at most between 2 concurrent hardware threads, this is the most efficient implementation for our purpuses. The shared pool is accessed by more than two threads, thus implemented using a lock-free stack.

4. EXPERIMENT

4.1 Thread Scheduler

Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks.

4.2 Concurrent Hash-Table

Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks.
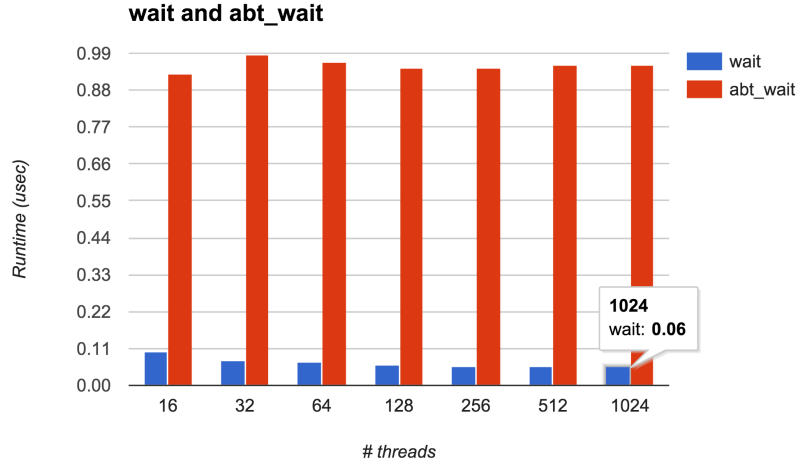
4.3 Overall overhead

Figure 2: Comparing Thread wait and signal with Argobots condition variable implementation. Runtime from entering wait until waken up by signal per thread.
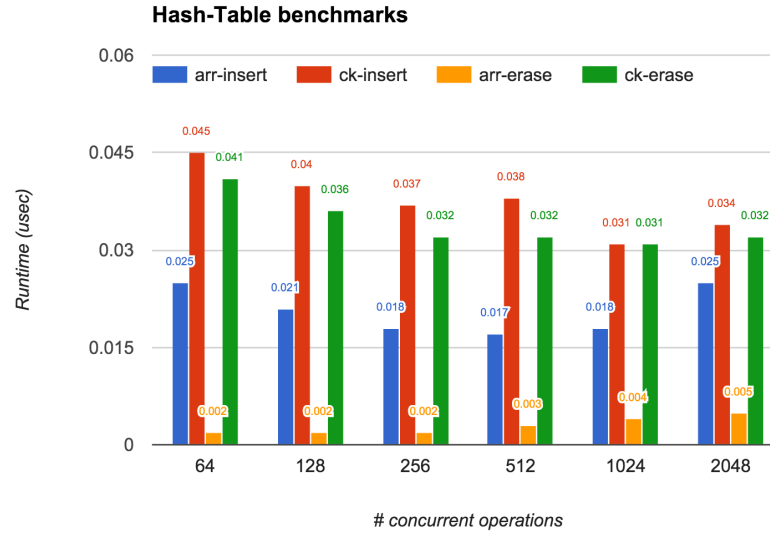

Figure 3: Comparing hash-table inserting and deletion by concurrent threads with cuckoo-hasing. Runtime is averaged per operation per thread. `arr` suffix is our implementation and `ck` prefix is cuckoo-hasing
.

Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks. Filling micro-benchmarks.

4.4 Pingpong multiple threads
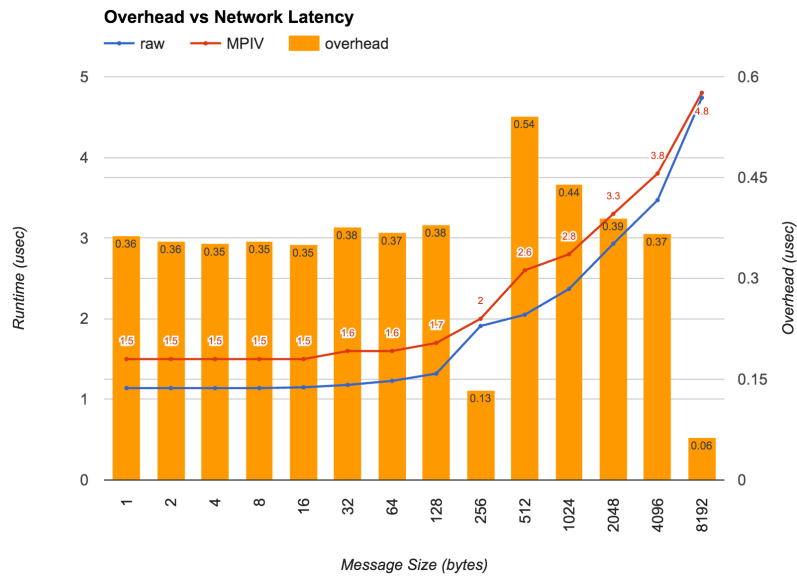
4.5 Applications

5. RELATED WORKS

6. CONCLUSION

Figure 4: Total overhead of runtime system, comparing to raw network latency.
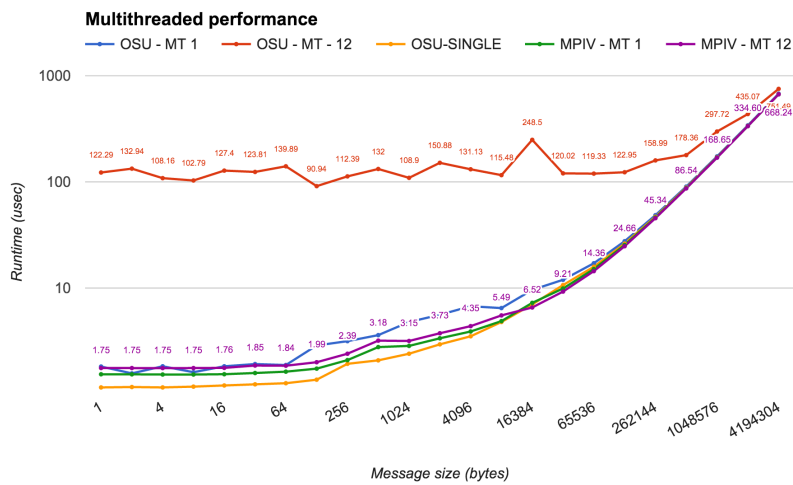

Figure 5: Comparing multi-threaded pingpong with OSU benchmarks using 1-thread in 1-worker and 12-threads in 12-workers. MPIV is our implementation.