Toward million communicating threads

Hoang-Vu Dang Marc Snir William Gropp

March 10, 2016

Abstract

We present a method for efficient runtime support of million concurrently communicating threads using Message Passing Interface.

1 Introduction

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 machine has their own implementation called CrayMPI, Intel also has a closed-source MPI implementation namely IntelMPI. Each of the implementation uses their best knowledge of the underlying computing machine and network architecture to optimize their code. This could improve performance in certain platform compared to the generalized approach in MPICH. 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 processor cores within a computing board, there raises a need of running MPI efficiently in a threaded environment. Although MPI requires all of its procedures must be thread-safe, the design of MPI implementation has never considered and optimized for using multiple threads. As a matter of fact, OpenMPI latest version still does provide a stable support of MPI_THREAD_MUTLIPLE i.e. no MPI procedure is allowed to be executed in concurrent thread. Other implementations such as 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. Although finer-grains

lock and concurrent hash-table allows MPI procedures to execute concurrently to certain extends, they when having a large number can adds additional overhead to the critical path and reduces performance of single-threaded applications. Moreover, communication in multiple threads could cause cache trashing or false sharing without a careful design. The processor can becomes under-ultiize because threads execute I/O operations can be preempted. Last but not least, kernel thread context-switching is even more costly than cache-misses. Several paper has report significant overhead for performing MPI concurrently in different thread.

For all of the above reasons, to our best knowledge there is no production application that uses the thread-safety feature of MPI although there is prediction that MPI+OpenMP is a solution to exascale and beyond. What has been ended up is the program runs MPI in a single-thread mode, and threads are used only for computation. This leads to a restrictive programming model similar to Bulk Synchronous Programming (BSP), in which communication and computation code appears in different phases. 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, non-transparent and inefficient. In this paper, we present an approximate implementation of MPI point-to-point communication which takes into account of multithreading 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 also provide a clear picture on the cost to support a complete MPI, in a finer-grain terms such as 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. The

implementation uses several queues data structure to store incoming 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 don't cares are not used.
- No concurrent send and receive having the same signature, so we do not have to 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) addressed. We also ignore, for the time being, one-sided operation.

We argue that these are reasonable assumption. Applications which relies on don't care or ordering could be rewritten to accomodate their needs. We shall discuss the possiblity in the later section. Moreover, relaxing the requirement allow a simple yet efficient implementation for communication in multiple threads. We shall discuss this in more detail in a later section.

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 op-

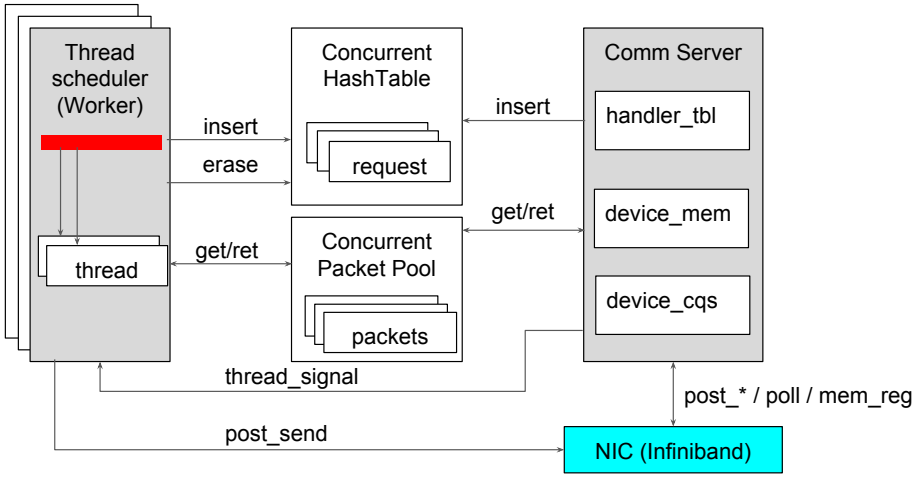

Figure 1: MPI Runtime Architecture for multi-threaded executions

timize 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 optimizing only a few neccessary operations.

Resource management is another issue in the runtime. 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 packet structure. The pool is the second shared data structure among all components of the network. 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 which allows a large concurrent threads to send and receive messages with a remote target. 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 value v' is returned, no changed are made to H. In constrat, the erase is always success, which shall replace any value stored with the input k with $\perp$, essentially this means 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 takes 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 safely use the hash-table to implement other concurrent objects.

The hash-table serves as a mechanism for a thread and the communication server to interact. We now describe algorithms for Send and Recv.

Message delivery in general can be implemented in two way: eager or rendevouz protocol. Eager protocol is when the message buffer is copied into an intermediate buffer - usually packed with other control data to deliver to the network. This allows the Send operation to return immediately since sent buffer can be reused. This protocol however becomes inefficient when message size gets larger, typically larger than the L2 or L3 cache size, when 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 thus saving extra copies. The protocol however requires some control messages to exchange meta data and signal completion.

2.2.1 Point-to-Point Eager protocol

Algorithm for eager protocol for a thread is listed in Algorithm 1. Algorithm for communication server when receiving a eager packet is

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:     Γ = current thread id.
10:    Create a request r = (b, s, d, Γ)
11:    Create hash-value v from r.
12:    Create hash-key k from d.
13:    v' = H.insert(k, v)
14:    if v' ≠ v then ▷ : insertion fail, message has arrived, copy the data.
15:        Copy v'.p.b to b
16:        pkpool.ret(p)
17:    else                    ▷ : insertion success, message has not arrived.
18:        ThreadWait()
19:    end if
20:    H.erase(k)
21: 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, thread has arrived, copy the data.
6:         Copy p.b to v'.r.b
7:         ThreadSignal(r.Γ)
8:         pkpool.ret(p)
9:     else     ▷ : insertion success, thread has not arrived, nothing to do.
10:        return
11:    end if
12: end procedure
```

listed in Algorithm 2. The basic idea here is that the thread and the communication server at the receiving side can effectively determine whether the message has arrived or whether there is a posted receiving request respectively.

If the communication server succeeds with the hash insertion, the receiving request has not been posted by a thread, 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 thread identification. The thread now block 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 thread identification. It now can find the thread and wake up 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 last communication 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 (cite here) and we refer the ready to that.

In our runtime, we can save one control message i.e. the RTS. The reason is we do not have wird-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 either waits when the RTR message has not arrived or when RDMA has not finished. In the first situation, the server will issue RDMA instead of the thread when RTR arrived. In both situation the server wake up the sender thread when RDMA has completed. On the other hand, the receiver only waits for the RDMA event to finish after issuing a eager-send of RTR.

2.3 Critical Operations Discussion

Clearly, optimizing the hash-table, packet pool and thread operations is the key to the performance of both protocol. One could achieve $O(1)$ amortized complexity. However, an efficient implementation requires optimizing them for the constant factor. Ideally, we want these operation to be *wait-free* and the constant factor is tiny. In the next section, we describe the implementaion of each of the component and show how we could optimize to reduce the cache misses in NUMA achitecture.

3 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 switch is designed to not involve OS system call, thus by-passing the kernel. This overall reduces the latency of switching to a new context to under a hundred cycle.

Rather than designing a general purpose ULT, we focus on the two most important operations: `ThreadWait` and `ThreadSignal`. One possible implementation is to use condition variable. This is the typical synchronization method which is used in both POSIX thread and 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.

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 concurrent thread in the system at a point in time. Additionally, during creation time, the worker atomically set its Γ bit

to 1.

Algorithm 3 Thread scheduler

```
1: procedure SCHEDULING(w, V)                          ▷ worker, bit-vector
2:     while !w.stop do                                ▷ loop until user ask to stop
3:         for word64 in V do
4:             if word64 ≠ 0 then
5:                 localWord64 = 0
6:                 ATOMIC_EXCHANGE(word64, localWord64)
7:                 while localWord64 > 0 do
8:                     b = Find-First-Set(localWord64)
9:                     FlipBit(localWord64, b)
10:                    ContextSwitch(Γ_b)
11:                end while
12:            end if
13:        end for
14:    end while
15: end procedure
```

Algorithm 4 Thread Operations

```
1: procedure THREADWAIT(sync)                          ▷ Synchronization object
2:     sync = (ω, Γ)
3:     ContextSwitch(ω)
4: end procedure
5:
6: procedure THREADSIGNAL(sync)                        ▷ Synchronization object
7:     w = sync.ω
8:     ATOMIC_SET_BIT(w.V, sync.Γ)
9: end procedure
```

Algorithms 3 describes our scheduling algorithms. The scheduler, when being free will look at each 64-bit word in the bit-vector to find a schedulable thread. Instead of working at 1 thread grenularity, we work at 64 threads within a 64-bit word. By using an atomic exchange to swap the interested word to local variable, we could continously perform read and write this variable without accessing the main memory. This is not only easier for correctness because there could be concurrent thread writing to the bit-vector, this also ensures progress property for all threads because we schedule all threads in a bit-vector before moving to the next word.

Algorithms 4 describes the two thread operations. `ThreadWait` simply stores the thread indentification into the synchronization object and switch back to the worker scheduler. On the other hand, `ThreadSignal` shall access the bit-vector and atomically set the bit in the appropriate word based on the thread identification 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. In practise, we store additionally an atomic flag per synchronization object to prevent this to happen, we ignore it for simplicity.

One can point out a problem with our thread scheduler design. The performance shall degrade greatly when we want to support a large number of threads. The reason is we have to iterate over the bit-vector. There could be an issue with fairness because thread with larger index will be scheduled later. 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 index and go directly to that word index to look for schedulable thread. 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* can happen because the secondary level bit is potentially shared by 64 thread of the first level bit. But this does not affect the correctness because the thread will not be scheduled twice in anyway. There is no situation that we can think of that we might have *false negative* or *true negative*.

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 8 words can fit in a cache line and better to read together, we can use 1 bit to represent a set of 8 64-bit words, and by having 8 words in the secondary level we can support 262144 concurrent threads per worker. Therefore, for a small number of worker, we can go up to our goal of million thread per node without sacrifying 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 take 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 and it allows 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 4-entry in case we have more than 3 collisions. Since 4-entry 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 to a special value indicating an empty entry. This is only possible because only one thread can change an entry key since no concurrent Send/Recv with the same tag is allowed. Thus our `empty` is very fast and achieves wait-freedom since no lock is required. Our `insert` is cache-friendly and does not typically suffer long conflicts when the table size is large enough.

3.3 Concurrent Packet Pool

Our pool is currently implemented using a lockfree stack. At the initialization, a fixed numbers of packet is allocated and push to the stack. During runtime a pool `get` is a stack `pop` and a pool `ret` is a stack `push`.

Since the packet is a unit of communication, its data is read and written very frequently. The stack implementation although does not give us the best latency in terms of concurrent accesses, it gives us good temporal locality of the packet data.

4 Experiment

4.1 Thread Scheduler

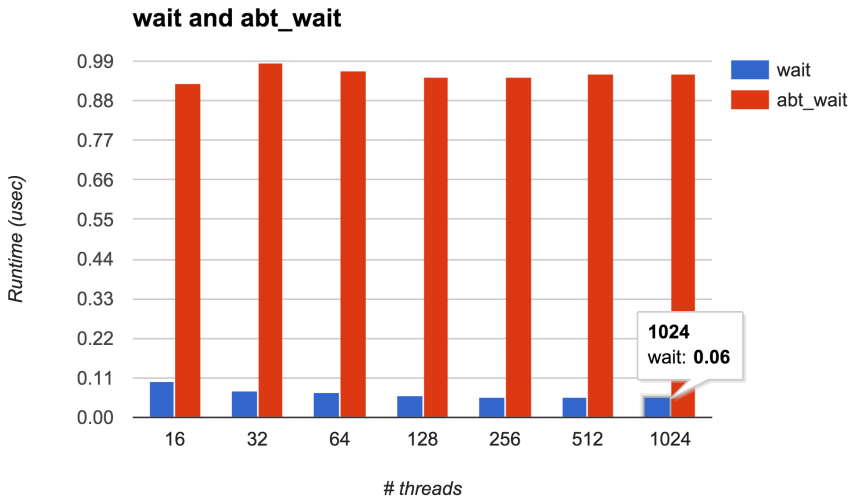

Figure 2: Comparing Thread wait and signal with Argobots condition variable implementation. Runtime from entering wait until waken up by signal per thread.

4.2 Concurrent Hash-Table

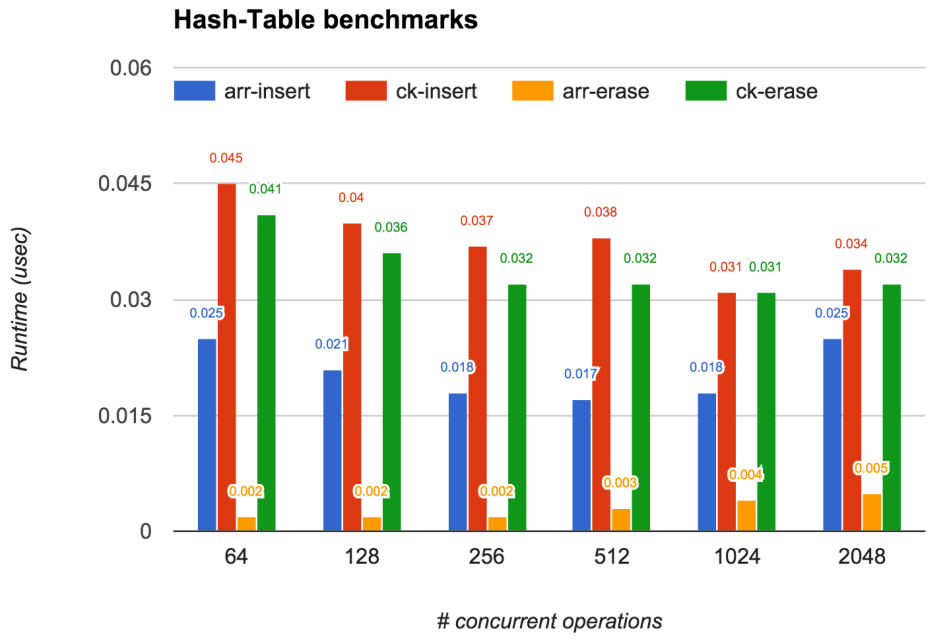

Figure 3: Comparing hash-table inserting and deletion by concurrent threads with cuckoo-hasing. Runtime is averaged per operation per thread. `arr` suffix is our implementation and `ck` prefix is cuckoo-hasing

.

4.3 Overall overhead

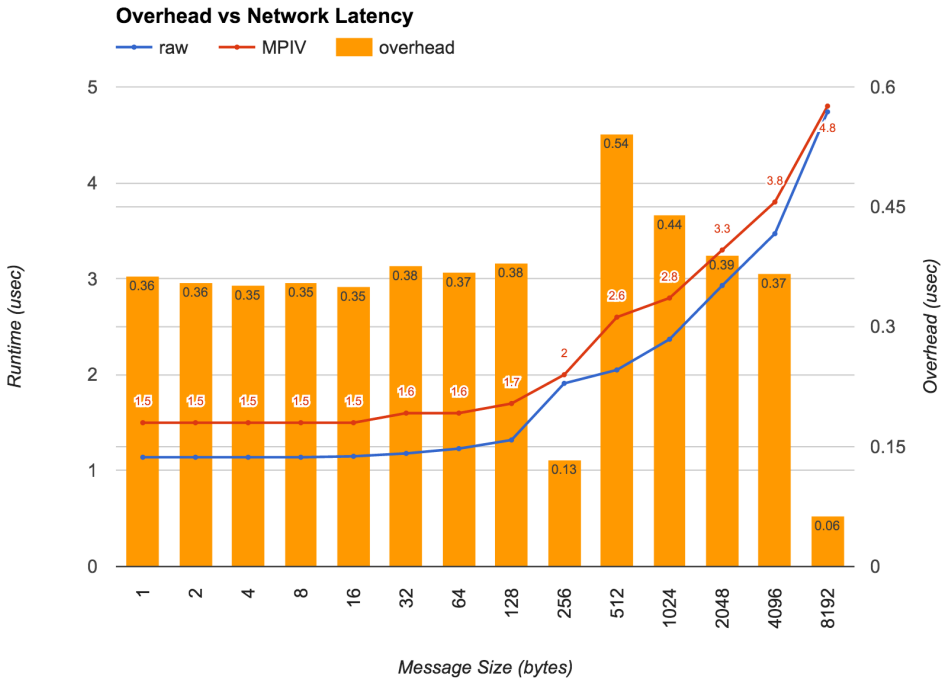

Figure 4: Total overhead of runtime system, comparing to raw network latency.

4.4 Pingpong multiple threads

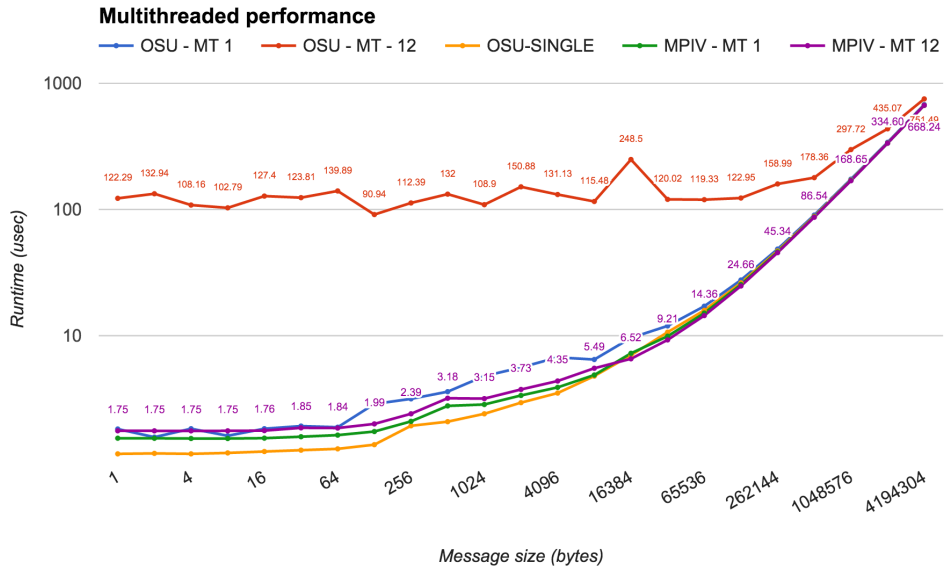

Figure 5: Comparing multi-threaded pingpong with OSU benchmarks using 1-thread in 1-worker and 12-threads in 12-workers. MPIV is our implementation.

4.5 Applications

5 Related Works

6 Conclusion