Toward million communicating threads

Hoang-Vu Dang   Marc Snir   William Gropp

February 24, 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 multi-threading 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 High level runtime architecture

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 will store message which has arrived but fail to find a matching send; the posted queue will store the pending requests which is waiting for incoming matched messages. Depending on different situation, 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-existing even though individual queue might be implemented as lock-free queue. Moreover, when the number of concurrent communication increases, traversing a queue requires linear time to the size of the queue.

Our first idea is to replace these queues with a single data structure: a concurrent hash-table. This is not a radical idea, however it has not been considered and implemented because of the following MPI semantics:

- MPI Send/Recv allows wird-card matching e.g. MPI_ANY_TAG, which could be used to match messages with any tag.
- MPI enforces an order of messages, two Sended messages having the same tag should arrive in the order that they are posted.

We argue that wild-card and ordering is not needed, any application which relies on those features could be rewritten slightly to accomodate their needs. 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.

We next tackle the problem of cache thrashing and long-delay context-switching by designing a message-driven runtime. 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 optimizing only a few neccessary operations.

2.1 Data Structure and 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. Our algorithms are based on the following assumptions about the user of MPI:

- There is no concurrent MPI_Send or MPI_Recv that having the same target i.e. either tag, source or communicator argument must be different.
- There is no use of wildcard - MPI_ANY_TAG, or MPI_ANY_-SOURCE.

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.

We next describe the algorithm for Eager and Rendevouz under our design.

2.2 Point-to-Point Eager protocol

Algorithm 1 Eager-message send/recv for thread

```
 1: procedure SEND-EAGER($b, s, d$)    ▷ : buffer, size, destination signature
 2:     Create new packet $p$
 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: procedure RECV-EAGER($b, s, d$)    ▷ : buffer, size, source signature
 8:     $\Gamma$ = current thread id.
 9:     Create a request $r = (b, s, d, \Gamma)$
10:     Create hash-value $v$ from $r$.
11:     Create hash-key $k$ from $d$.
12:     $v' = H.\text{insert}(k, v)$
13:     if $v' \neq v$ then    ▷ : insertion fail, message has arrived.
14:         Obtain request $r$ from $v'$
15:     else    ▷ : insertion success, message has not arrived.
16:         ThreadWait()
17:     end if
18:     Copy $r.b$ to $b$.
19:     $H.\text{erase}(k)$
20: end procedure
```

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

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, thread has arrived.
 6:         Obtain request $r$ from $v'$
 7:         ThreadSignal($r.\Gamma$)
 8:     else                         ▷ : insertion success, thread has not arrived.
 9:         return
10:     end if
11: end procedure
```

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.

2.3 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.

2.4 Thread and hash-table operations

Clearly, optimizing the hash-table and thread operations is the key to the performance of both protocol. One could easily achieve $O(1)$ amortized complexity for both. 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 fact, in the next section we show that we can achieve wait-free for all but the hash-table insert operation. Moreover, our implementation in avarage costs about 1 cache miss for each operation. Note that one can consider implement thread operations using using condition variable. However, these are not wait-free since condition variable in general requires a lock. A busy-waiting implementation is even worst since the processor spends useless time polling.

3 Implementation and Optimization

4 Experiment

5 Related Works

6 Conclusion