Towards millions of communicating threads

Hoang-Vu Dang, Marc Snir, William Gropp
University of Illinois at Urbana-Champaign (UIUC)
Motivation

• Machine node with many cores inter-connected with Intelligent NIC
• Programming model: MPI+X
   – X: intra-node thread parallelism (e.g. OpenMP); can use lightweight threads (tasks)
• MPI performance does not scale with many threads
Motivation

MPI+X Performance Issues

• Semantics of message matching, implemented using 2-queue data structure
• Thread scheduler is not aware of status of communications
• Low-level communication layer is not designed for multi-core architecture because of many mutual-exclusion point
Our bottom-up approach

• Re-design the message matching mechanism for multi-threading
• Design an efficient communication interface between MPI library and thread scheduler
• Carefully manage resources to avoid mutual exclusion of many threads and optimize for cache locality
MPI point-to-point message-matching
MPI point-to-point message-matching

RECV issuer comes first:
1. SEARCH(UQ)
2. PUSH(PQ)
3. SEARCH(PQ)
4. POP(PQ)

MSG arrives first:
1. SEARCH(PQ)
2. PUSH(UQ)
3. SEARCH(UQ)
4. POP(UQ)
MPI point-to-point message-matching

- Multi-threaded scenario

**Each thread is delayed by O(M*N)**

M: # pending messages
N: # number of threads
MPI point-to-point message-matching

Experiment is done on Stampede Cluster: 16 core – FDR Infiniband

OSU multi-threaded latency benchmarks

Ideal case: multi-threaded performance should remain the same as single threaded

>100x
MPI point-to-point message-matching

- No mixing of wildcard and normal match
MPI point-to-point message-matching

• Specialized hash-table with one operation: \texttt{ACCESS(K,V)}:
  – found then remove and return value
  – otherwise insert into the table

• Symmetric notion of message matching:
  – IF \((V' = H.\texttt{ACCESS}(K,V)) == \perp\) 
    • No unexpected packet / No posted request
  – ELSE 
    • \(V'\) is the matched entry (packet/request)
    • \(V'\) is removed from the hash table
MPI point-to-point message-matching

• Specialized hash-table with one operation: ACCESS(K,V):
  – found then remove and return value
  – otherwise insert into the table

• Symmetric notion of message matching:
  – IF ((V’ = H.ACCESS(K,V)) == ⊥ )
    • No unexpected packet / No posted request
  – ELSE
    • V’ is the matched entry (packet/request)
    • V’ is removed from the hash table
MPI thread scheduler

- What to do when a thread cannot complete its blocking request?
  - Current approach: randomly yield to other
- Better approach: communication-aware
  - de-scheduled when cannot complete
  - dedicates one or more core for communication progress; wake up thread when finished.
Proposed runtime architecture

- Thread scheduler
- Concurrent HashTable
- Concurrent Packet Pool
- Scheduling table
- Communication Server
- NIC
Algorithm for receive matching (eager send)

THREAD: MPI_RECV
• V : Request
• V' : Packet
• IF ((V' = H.ACCESS(K,V)) == )
  – WAIT()
• ELSE
  – Finish and return

SERVER: POLL_CQ
• V : Packet
• V' : Request
• IF ((V' = H.ACCESS(K,V)) == )
  – Continue
• ELSE
  – SIGNAL()
Algorithm for receive matching (eager send)

**THREAD: MPI_RECV**
- V: Request
- V': Packet
- IF ((V' = H.ACCESS(K,V)) == ⊥)
  - WAIT()
- ELSE
  - Finish and return

**SERVER: POLL_CQ**
- V: Packet
- V': Request
- IF ((V' = H.ACCESS(K,V)) == ⊥)
  - Continue
- ELSE
  - SIGNAL()

How to implement these efficiently?
MPI Threading Interface

• Require two critical operations over a synchronization object
  – SIGNAL
  – WAIT

• Associate a request/thread with a synchronization object
  – Communication server find the object inside request and signal the waiting thread.

• Semaphore / condition variable
  – Implemented using waiting queue
MPI Threading Interface: PThread

The lower the better

![Graph showing OSU multi-threaded latency (64 bytes)](image)

- MVAPICH2
- PTHREAD
- Ideal

- 3x improvement for MVAPICH2 compared to PTHREAD
MPI Threading Interface: Argobots

The lower the better

Can we do better than this?
MPI Threading Interface: FULT

• Key idea: using bit-vector for ready threads structure
  – 0: executing or blocked
  – 1: runnable
• SIGNAL: atomic bit-set instruction
• WAIT: context-switch to other runnable threads (find-first-bit-set instruction)
MPI Threading Interface: FULT

The lower the better

OSU multi-threaded latency

Latency (usec)

# Threads

1

2

4

8

16

PTHREAD

ABT

Ideal

FULT

1.2x
Summary: Initial assumption

A. MPI point-to-point:
   1. No wildcard
   2. No pending requests with the same signature <communicator, rank, tag>

B. Integrated with user-level threads (ULT)
   1. No thread migration
   2. No fairness requirement

C. Based on NIC with modern features:
   1. Communication Server
   2. RDMA, address translation is done in hardware
Runtime optimization and implementation

• Concurrent Hash Table: ACCESS
  – Open Hashing with Fat entry for cache locality
  – Per bucket spinlock is sufficient

• Packet Pool: ALLOC/FREE
  – Locality-aware: consumer will perform memory copy
  – NUMA-aware: pool per core, stealing for balancing

• Thread scheduler (FULT): WAIT/SIGNAL/YIELD
  – Two-level of bit-vector for large number of threads
  – YIELD == Self Signal then Wait
Scaling to 1M threads (over-decomposition)

The lower the better
## Communication Kernel
Tested implementation

<table>
<thead>
<tr>
<th>Notation</th>
<th>MPI</th>
<th>Scheduler</th>
<th>Core assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>mvapich2</td>
<td>MVAPICH2 (2.1)</td>
<td>POSIX thread</td>
<td>1 core per rank</td>
</tr>
<tr>
<td>mvapich2+async</td>
<td>MVAPICH2 (2.1)</td>
<td>POSIX thread</td>
<td>2 cores per rank</td>
</tr>
<tr>
<td>pthread+hash</td>
<td>customized</td>
<td>POSIX thread</td>
<td>16 cores per rank</td>
</tr>
<tr>
<td>abt+hash</td>
<td>customized</td>
<td>ULT (Argobots)</td>
<td>16 cores per rank</td>
</tr>
<tr>
<td>fult+hash</td>
<td>customized</td>
<td>ULT (Fult)</td>
<td>16 cores per rank</td>
</tr>
</tbody>
</table>

- NAS Data Traffic:
  - Single-threaded performance with three different communication patterns.
- Unbalanced Tree Search:
  - Distributed work-stealing for tree traversal
- Graph500:
  - Breadth-First-Search over large distributed graph
NAS Data Traffic

- **BH**: black hole – multiple sender, one receiver
- **WH**: white hole – one sender, multiple receivers
- **SH**: shuffle – all-to-all like communication
- 128 MPI rank, one per compute node.

The higher the better
Unbalanced Tree Search

- Distributed memory Work-Stealing implementation
- ~ 3M node binomial tree (T3XXL)
- Compare single-threaded ref. code with using threads for receiving work-stealing requests
Graph 500

- Multi-threaded by spawning threads to receive from different targets
- Weak scaling up to 4096 cores at graph scale-28.
- Compare ref. code with using threads for receiving vertices from multiple nodes.
Conclusion

- We have designed and implemented low-level MPI communication with a large number of threads.
- Our techniques include:
  - relaxation of wildcard semantics
  - tightly-coupled design of MPI and thread scheduler
  - resource management for cache locality
Future works

• MPI:
  – Waitany, Waitall, Waitsome...

• Thread scheduler:
  – Fairness and thread migration

• Low-level interface:
  – NIC offloading (Omnipath)

• Applications & Benchmarks
Related works


Acknowledgement

• Alex Brooks, Nikoli Dryden (UIUC)
• Ron Brightwell (SNL)
• Pavan Balaji (ANL)

Thank you,
Question please!
Contact

- Hoang-Vu Dang: hdang8@illinois.edu