Parallelized Software Offloading of Low-Level Communication with User-Level Threads

Wataru Endo, Kenjiro Taura

Graduate School of Information Science and Technology
The University of Tokyo

January 31, 2018

Summary

- We implemented parallelized software offloading with user-level threads for accelerating HPC interconnects
  - Especially on multi-threading environments
- Our offloading method achieves:
  1. **Parallelized** communications
     - The aggregated message rate is 4x larger than serialized one
  2. Preserve the benefits of software offloading
     - **Latency hiding**, **multi-threading** performance
  3. **Reduced consumption of CPU resources**

Introduction (1)

- HPC interconnects provide high-performance communication among nodes
  - e.g. InfiniBand, Omni-Path, uGNI, Tofu
- We focused on improving the communication performance on **InfiniBand**:
  - One of the most widely used interconnection networks in HPC
  - Supported operations:
    - Two-sided communication (SEND/RECV)
    - One-sided communication (RDMA WRITE, READ, atomic)

Introduction (2)

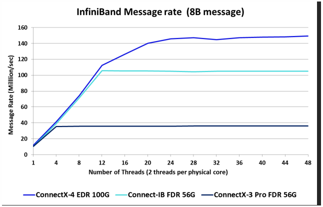
- The performance of HPC interconnects is often limited by the software overhead
  - Issuing a message takes hundreds of cycles in CPUs
  - Application threads cannot proceed during this overhead
    - Prohibits latency hiding
  - Message rate from a core is limited to $\approx$ 10 million/sec
    - The latest hardware can provide $>$ 100 million/sec

[1] https://www.openfabrics.org/images/eventpresos/workshops2015/UGWorkshop/Friday/friday_01.pdf

Introduction (2)

- The performance of HPC interconnects is often limited by the software overhead
  - Issuing a message takes hundreds of cycles in CPUs
  - Application threads cannot proceed during this overhead
    - Prohibits latency hiding
  - Message rate from a core is limited to $\approx$ 10 million/sec
    - The latest hardware can provide $>$ 100 million/sec

- **Multi-core** processing is required
  - Network software stacks should be able to efficiently operate in multi-threading

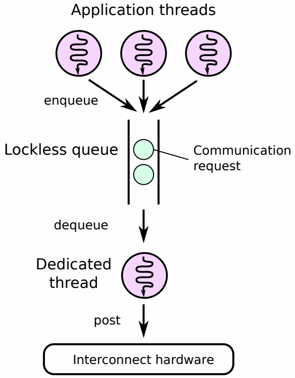


[Mellanox Technologies ’15] [1]

[1] https://www.openfabrics.org/images/eventpresos/workshops2015/UGWorkshop/Friday/friday_01.pdf

Introduction (3)

- We are focusing on **software offloading**:
  - Use **dedicated threads for communication**
  - Delegate the communication processing via lockless queues
- Benefits of software offloading:
  - Improves message rates
  - Reduces message injection overheads
- Example: Software offloading in MPI [Vaidyanathan et al. '15]:
  - Set the underlying MPI runtime to a single-threaded mode (MPI_THREAD_SERIALIZED)
  - Only one thread handles actual MPI communication



Introduction (4)

- Software offloading has disadvantages
  1. Latency is increased
  2. CPU resources are consumed in vain

Introduction (4)

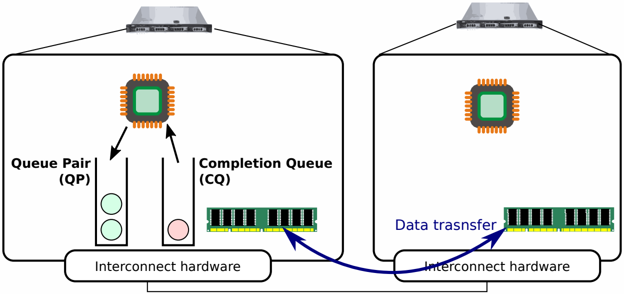
- Software offloading has disadvantages
  1. Latency is increased
  2. CPU resources are consumed in vain
- Example: **PAMI** [Kumar et al. '13]
  - Implements an offloading method as a low-level communication library
  - Can start & stop the offloading threads using a special feature of POWER8 processor

Introduction (4)

- Software offloading has disadvantages
  1. Latency is increased
  2. CPU resources are consumed in vain
- Example: **PAMI** [Kumar et al. '13]
  - Implements an offloading method as a low-level communication library
  - Can start & stop the offloading threads using a special feature of POWER8 processor
- We provide a method to **dynamically start & stop** the offloading threads
  - Using a **user-level thread (ULT)** library

Background: InfiniBand

- **Queue Pair (QP)**
  - A hardware queue to which new requests are posted
- **Completion Queue (CQ)**
  - A hardware queue that notifies the completion of communication
- InfiniBand Verbs: A standard low-level API for InfiniBand
  - Provides thread-safety of all the API functions
  - Each QP and CQ are guarded by a **spinlock**



Background: Portable low-level communication libraries

- InfiniBand Verbs is too low-level for building the system software
  - Portable low-level communication libraries, which work on different interconnection mechanisms, are required
- Existing portable low-level communication libraries
  - Relatively old libraries:
    GASNet [Bonachea et al. '02], ARMCI [Nieplocha et al. '06]
  - Recent libraries (2014-):
    UCX [Shamis et al. '15], libfabric [Grun et al. '15],
    ComEx [Daily et al. '14]
- The motivation of these libraries is the portability

Background: Multi-threading of MPI

- MPI+X is recently discussed
  - "X" corresponds to shared-memory systems (e.g. OpenMP)
- **MPI_THREAD_MULTIPLE** guarantees thread safety
  - The MPI implementation is supposed to work efficiently in multi-threading programs
  - Still immature in most of MPI implementations (e.g. [Balaji et al. '10])

Background: Multi-threading of MPI

- MPI+X is recently discussed
  - "X" corresponds to shared-memory systems (e.g. OpenMP)
- **MPI_THREAD_MULTIPLE** guarantees thread safety
  - The MPI implementation is supposed to work efficiently in multi-threading programs
  - Still immature in most of MPI implementations (e.g. [Balaji et al. '10])
- **MPI Endpoints** [Dinan et al. '13]
  - Provide additional ranks per process
  - Select endpoints manually by MPI users
  - Each endpoint has its own set of QPs & CQs
    - Needless to manage thread safety inside endpoint functions
  - Not standardized, but may be included in MPI-4

Background: Sharing Endpoints

- A remaining question: How many endpoints should be created?
  - Too few endpoints cannot exploit the parallelism of a NIC
  - Too many endpoints increase cache misses in a NIC
    - Assume one-to-one connections in N nodes with M-core CPUs
    - # of necessary QPs is NM^2 per node

Background: Sharing Endpoints

- A remaining question: How many endpoints should be created?
  - Too few endpoints cannot exploit the parallelism of a NIC
  - Too many endpoints increase cache misses in a NIC
    - Assume one-to-one connections in N nodes with M-core CPUs
    - # of necessary QPs is NM^2 per node
- Solutions
  1. Avoid using "Connected" protocols (e.g. [Kalia et al. '16])
     - For the UD protocol, it's unnecessary to share QPs because a "Datagram" QP can send messages to multiple QPs
     - However, UD cannot support RDMA operations
  2. **Share a moderate number of QPs & CQs**
     - We chose this solution to use RDMA
- Software offloading can mitigate the drawback of sharing QPs
  - Because it can reduce the resource contentions using atomic operations

Background: User-Level Threads + Communication

- Parallelizing communication & software offloading
  - Multiple spinning threads consume too many CPU resources
  - We adopted user-level threads to implement starting & stopping them

- **User-Level Thread (ULT)** libraries are recently becoming popular
  - Efficiently schedule “threads of executions” in user space

```
ult_id_t ult_fork(
  void (*)(void*), void*);
void ult_join(ult_id_t);
void ult_detach(ult_id_t);
void ult_yield();
```

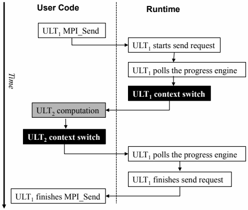
Background: User-Level Threads + Communication

- Parallelizing communication & software offloading
  - Multiple spinning threads consume too many CPU resources
  - We adopted user-level threads to implement starting & stopping them

- **User-Level Thread (ULT)** libraries are recently becoming popular
  - Efficiently schedule “threads of executions” in user space

```
ult_id_t ult_fork(
  void (*)(void*), void*);
void ult_join(ult_id_t);
void ult_detach(ult_id_t);
void ult_yield();
```

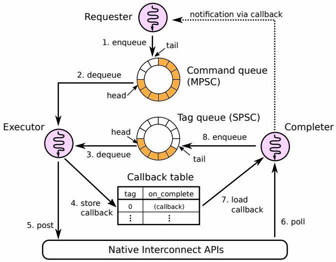
- Example: MPI+ULT [Lu et al. ’15]
  - Latency hiding of MPI applications using user-level threading
  - Their method did not utilize the parallelism of network resources



[Lu et al. ’15]

Implementation

- 2 types of queues:
  - **Command queues** hold new communication requests
  - **Tag queues** hold the tags attached to the ongoing requests
- 3 types of components (threads):
  - **Requesters** are the application threads inserting communication requests to the command queue
  - **Executors** monitor the command queue and post the communication requests to the hardware
  - **Completers** poll the completion of communication



Offloading architecture

Implementation: How to Keep Awake

- Problem:
  How to guarantee that the communication threads are NOT sleeping when there are ongoing requests?
  - There may be a race condition if
    1. The queue's producer considers the consumer is awake
    2. The queue's consumer starts sleeping

Implementation: How to Keep Awake

- Problem:
  How to guarantee that the communication threads are NOT sleeping when there are ongoing requests?
  - There may be a race condition if
    1. The queue's producer considers the consumer is awake
    2. The queue's consumer starts sleeping
- Solutions
  1. Mutexes + condition variables
     - A standard solution for this problem
     - Suffers from the overhead of system calls
  2. **Atomic operations + user-level threads**
     - Minimizes the overhead to synchronize between threads
     - **Embed a bit whether the consumer is sleeping or not in the queue's counter**
     - If sleeping, awake the consumer using user-level threads

Implementation: Requesters

- Procedure of requesters:
  1. Select one endpoint from the set of avaiable endpoints
     - We used a thread-local storage to select in a round-robin fashion
  2. Do a compare-and-swap (CAS) to `tail` of the command queue
     - Replace [count, 0 or 1] with [count+1, 1]
     - Synchronize both with the consumer (= executor) and other requesters
  3. Place a new command on the buffer
  4. If the old `tail`'s LSB was 0 (= sleeping),
     fork a new user-level thread to resume the executor

Implementation: Executors

- Procedure of executors:
  1. Get a command and a tag from the queues
  2. Post a communication request to the hardware
     - Call `ibv_post_send()` in InfiniBand
  3. Recycle the command entry
  4. If the completer is sleeping, fork a new user-level thread for the completer
  5. If there is no request in the command queue, try to start sleeping
     - Reset the LSB of the command queue's `tail` using a CAS

Implementation: Completers

- Procedure of completers:
  1. Poll a completion from the hardware
     - Call `ibv_poll_cq()` in InfiniBand
  2. Invoke the callback function
  3. Recycle the tag
  4. If there is no ongoing request, try to sleep
  5. If there are ongoing requests, but polling failed, then call `ult_yield()`

Evaluation

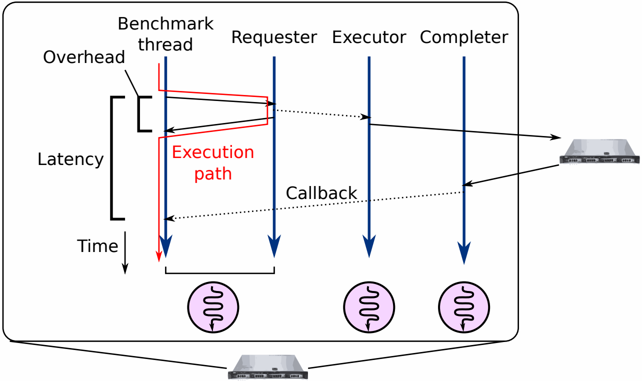
- Microbenchmark on these metrics:
  - Latency, overhead, and message rate
- Runs 2 processes (1 process/node)
  - One process has benchmark threads repeating RDMA READ
- MassiveThreads 0.97 as a ULT system
  - Change to use parent-first scheduling (child-first is the default)
  - Run only 10 worker threads/node to avoid NUMA effects

Evaluation Environment

CPU	Intel® Xeon® E5-2680 v2 2.80GHz, 2 sockets$\times$ 10 cores/node
Memory	16GB/node
Interconnect	Mellanox® Connect-IB® dual port InfiniBand FDR 2-port (only 1 port is used)
Driver	Mellanox® OFED 2.4-1.0.4
OS	Red Hat® Enterprise Linux® Server release 6.5 (Santiago)
Compiler	GCC 4.4.7 (with the option "-O3")

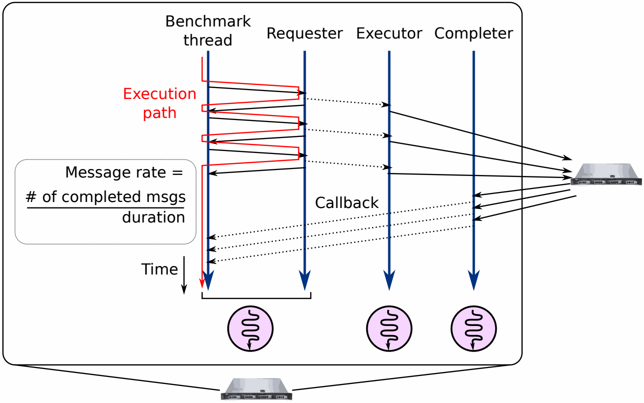
Evaluation: Measuring Latency & Overhead

- Microbenchmark to measure latency & overhead
  1. Each thread makes a new communication request
  2. Waits for its completion by spinning on the flag
     - Call `ult_yield()` if it's not completed
  3. The callback function will set the flag



Evaluation: Measuring Message Rates

- Microbenchmark to measure a message rate:
  1. Each thread continuously makes communication requests until # of ongoing requests reaches the batch size (= 256 in this paper)
  2. The callback function will fetch-and-add the counter
  3. After 5 seconds, read the counter and calculate the rate

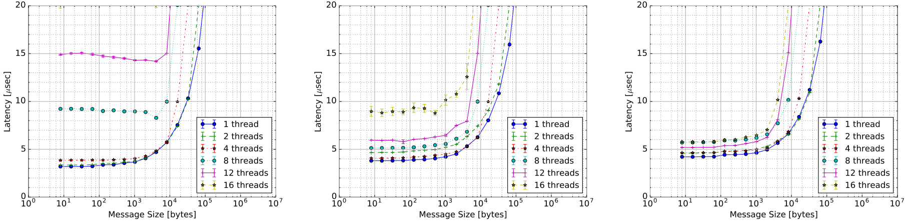


Evaluation: Methods

- Compare 3 different methods:
  - **Direct injection**
    - The post function is directly called in application threads
    - The polling thread (= completer) is executed in a different thread
    - Shared resources are guarded by spinlocks
  - **Static offloading**
    - There is an executor thread that is spinning on a commmand queue
    - Typical software offloading approaches
  - **Dynamic offloading**
    - An executor thread is dynamically spawned from application threads

Evaluation Results: Latency with 1 QP & CQ

- Reference: 2.01 μsec in perftest benchmark
- 3.197 μsec in Direct injection
  - Overhead of separating polling threads
- 3.804μsec in Static offloading
  - Overhead of sending a request to an executor thread
- 4.21μsec in Dynamic offloading
  - Overhead of waking up an executor & completer thread

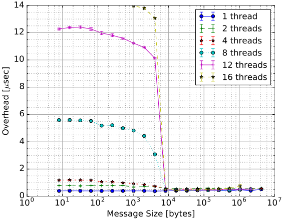


Direct injection | Static offloading | Dynamic offloading

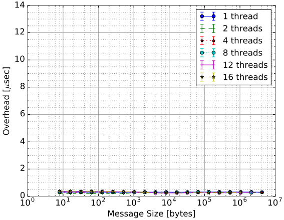
Horizontal axis: message size. Vertical axis: round-trip latency.
Each line represents # of requester threads.

Evaluation Results: Overhead with 1 QP & CQ

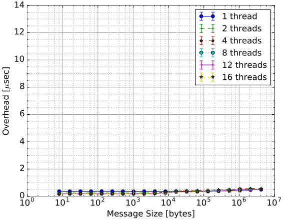
- Direct injection increases the overhead with ≥ 8 threads
  - Due to spinlock contentions
- Both static offloading & dynamic offloading can lower the overhead
  - Lockless queues reduce contentions



Direct injection



Static offloading

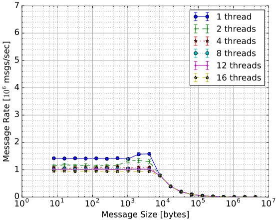


Dynamic offloading

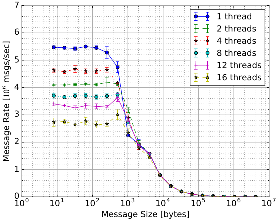
Horizontal axis: message size. Vertical axis: overhead of message injection.
Each line represents # of requester threads.

Evaluation Results: Message Rate with 1 QP & CQ

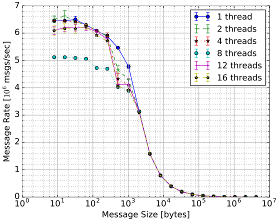
- 1.404 million/sec in Direct injection
  - 0.712 μsec per message, mostly coming from a Verbs function call
- 5.47 million/sec in Static offloading
  - Aggregating messages improved the message rate because of the usage of PCIe DMA [Kalia et al. '16]
- 6.452 million/sec in Dynamic offloading
  - The reason why it performs better than Static offloading is unknown



Direct injection



Static offloading



Dynamic offloading

Horizontal axis: message size. Vertical axis: message rate.

Each line represents # of requester threads.

Evaluation: Message Rate with Multiple QPs & CQs

- Use multiple QPs and CQs in Dynamic offloading:
  - Allocate a distinct CQ for each QP
  - Run an executor (user-level) thread for each QP
  - Run an completer (user-level) thread for each CQ
- Compare 2 methods:
  - “Fork”: Fork a new (user-level) thread when necessary

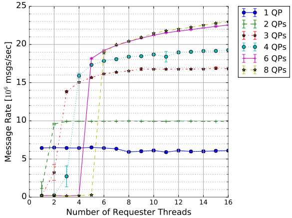
```
if (/* need to wake up a thread*/) {
    ult_id_t id = ult_fork();
    ult_detach(id);
}
```

  - “Condition variables”: Use a condition variable of MassiveThreads

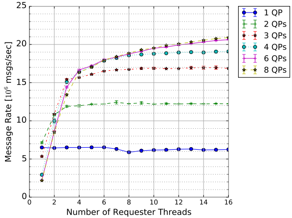
```
if (/* need to wake up a thread */) {
    myth_mutex_lock(&mtx);
    myth_cond_signal(&cv);
    myth_mutex_unlock(&mtx);
}
```

Evaluation Results: Message Rate with Multiple QPs & CQs

- The aggregated message rate increased to about 20 million/sec
  - With more QPs & CQs up to 6
- Highly degraded with a few QPs & requester threads
  - Workers are out of resources in “Fork”
  - Additional synchronizations in “Condition variables”



Fork (parent-first)

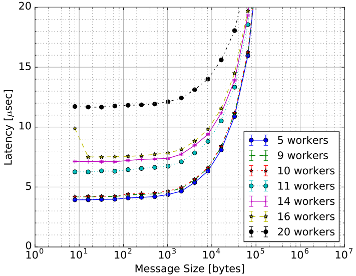


Condition variables

Horizontal axis: # of requester threads. Vertical axis: message rate.
Each line represents # of QPs & CQs.

Evaluation Results: NUMA Effects on Latency

- $\leq$ 10 workers, the latency doesn't change ($\approx 4\mu$sec)
- $>$ 10 workers, the latency jumps to 6.268μsec
  - Could not fit in one NUMA domain
  - When a worker of MassiveThreads starts stealing, it uniformly selects the worker at random
  - Offloaded communications may suffer from NUMA communication costs
- Not easy to solve this problem in general
  - Locality-aware work-stealing methods may improve the performance of our offloading system



Horizontal axis: message size.
Vertical axis: round-trip latency.
Each line represents # of worker threads.

Conclusions

- A parallelized software offloading scheme of low-level communication with user-level threads
  - Software offloading without busy-waiting
  - Reduced message injection overheads
  - Better aggregated message rates
- Future work
  - Use real applications for evaluation
  - Compare the performance with other libraries in detail
  - Shrink the latency increase in NUMA environments

Acknowledgements

- Information Technology Center (ITC), the University of Tokyo
- Assoc. Prof. Toshihiro Hanawa, the University of Tokyo