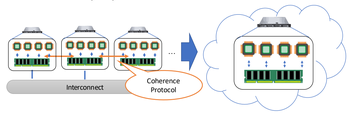
A Distributed Shared Memory Library with Global-View Tasks on High-Performance Interconnects

Wataru Endo, Kenjiro Taura (Graduate School of Information Science and Technology, The University of Tokyo)

Introduction

- Goal of our research:
  - Improve the productivity & performance of applications on distributed-memory machines
- Our central idea:
  shared memory + task parallelism
  - Two general programming models applicable to arbitrary parallel computation patterns
  - Intuitive global-view programming

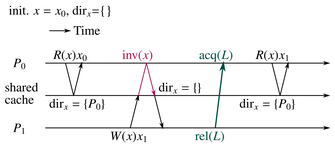
- **Distributed Shared Memory (DSM)**
  - **Physically distributed, virtually shared**
  - The system automatically synchronizes the caches between cores



- History of DSM
  - 1990s: Early DSM systems appeared
    - e.g. TreadMarks [Keleher et al. '94], JIAJIA [Hu et al. '98]
  - 2000s-: **PGAS** systems replaced them
    - e.g. UPC [El-Ghazawi et al. '02], Global Arrays [Nieplocha et al. '06], OpenSHMEM [Chapman et al. '10]
    - Scalable & global-view programming models
    - Explicit communications are still burdensome
- Why DSM again?
  1. Improvement of network speed [Ramesh '13]
     - Inter-node latency / DRAM latency ≈ 1000 (1990s), 10 times (2010s)
     - Inter-node bandwidth / DRAM bandwidth ≈ 500 (1990s), 2.5 times (2010s)
  2. Relationship with **many-core** architectures
     - Shared memory is considered as a bottleneck of scalability
     - Techniques for software DSMs are revisited

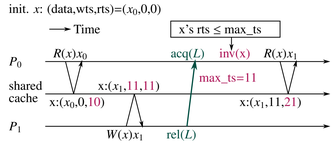
Cache invalidation methods

- Directory-based coherence
  - The state-of-the-art method to implement large-scale shared memory
  - Tracking sharers in centralized directories



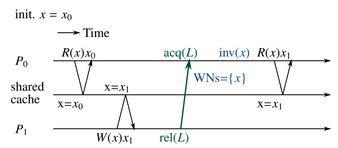
- Problems of directories:
  1. $O(P)$ storage cost to hold sharers (P: # of nodes)
  2. **Communication traffic of small invalidation messages**
  3. Complex state management leads to system bugs

- **Logical-timestamp-based coherence** [Yu et al. '15]
  - Invalidate cache blocks based on logical timestamps (= Lamport clocks)



- Pros of logical-timestamp-based coherence
  - Only $O(\log P)$ storage is required
  - No explicit invalidation message is needed
- Cons of logical-timestamp-based coherence
  - Unnecessary cache invalidations (= cache misses) due to the nature of logical timestamps

- **Write notice** (WN) (in TreadMarks [Keleher et al. '94])
  - Transfer a set of IDs of written cache blocks on each synchronization



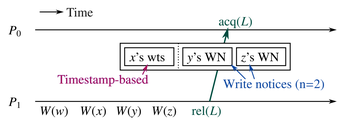
- Pros of write notices
  - # of invalidation messages is reduced
  - Unnecessary cache misses don't increase
- Cons of write notices
  - # of write notices infinitely grow during execution; TreadMarks used a complex global garbage collection mechanism

Implementation of our DSM library

- Numerous design options for DSM systems
- The ideas borrowed from ArgoDSM [Kaxiras et al. '15]:
  - Relaxed consistency model (assuming data-race-free programs)
  - RDMA-based implementation
  - Page-based DSM (vs. compiler-based)
  - Multiple-writer home-based eager protocol

Additionally, we developed 3 techniques:

1. **Hybrid cache invalidation method** of both logical-timestamp-based coherence and write notices
   - Enables to balance storage costs and # of cache misses



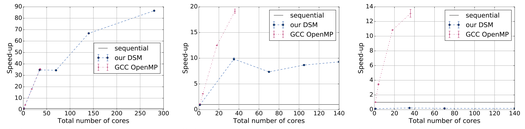
2. **Migrating home-based protocol**
   - "Always" migrates the home to the latest writer
   - Reduces the write latency when the same process writes again
   - The latest home is searched via "probable owners" [Li et al. '89]
3. **Call stack management over the DSM**
   - Simplifies global-view task migration (not evaluated in this poster)
   - Allows accessing the automatic variables of other threads; shared-memory programs can transparently work

Evaluation of NAS Parallel Benchmark on DSM

- Implemented a DSM library & an OpenMP wrapper on MPI
  - Currently, only static scheduling & non-nested loops are supported
  - Some features including reductions are not supported

CPU	Intel® Xeon® E5-2695 v4 2.1 GHz (max. 3.3 GHz with Turbo boost) 18 cores × 2 sockets / node
Memory	256GB / node
Interconnect	Mellanox® Connect-IB® dual port InfiniBand EDR 4x
OS	Red Hat® Enterprise Linux® 7.2
Compiler	GCC 4.8.5 (with the option "-O3")
MPI	MVAPICH 2.2

- Speed-ups of NAS Parallel Benchmark
  - Strictly speaking, we used an unofficial OpenMP C version [1]
  - Parallel reductions are replaced with serial loops



(a) NAS EP (CLASS=C) (b) NAS CG (CLASS=C) (c) NAS BT (CLASS=A)

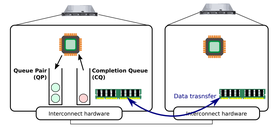
- Only NAS EP (Embarrassingly-Parallel) on our DSM becomes faster than in default OpenMP implementation
- Ongoing efforts for performance improvements
  - e.g. multi-threading communication performance, prefetching

Communication library for DSM

- We also implemented a communication library designed mainly for DSM (or PGAS) systems
  - Such systems tend to require **fine-grained** communications
  - Current CPU & interconnect architectures require **multi-threaded** communications to achieve the maximum performance
  - Traditional communication libraries are optimized for coarse-grained & single-threaded communications

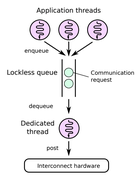
We assumed InfiniBand as the underlying interconnect:

- Queue Pair (QP)
  - A hardware queue to which new requests are posted
- Completion Queue (CQ)
  - A hardware queue that notifies the completion of communication



Software offloading

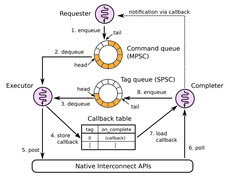
- We focused on **software offloading** [Vaidyanathan et al. '15]
  - Use **dedicated threads for communication**
  - Delegate the communication processing via lockless queues



- Pros of software offloading:
  1. Improves message rates
  2. Reduces message injection overheads
- Cons of software offloading:
  1. Latency is increased
  2. CPU resources are consumed in vain
- Example: **PAMI** [Kumar et al. '13]
  - Can start & stop the offloading threads using a special feature of POWER8 processor
- We provided a method to **dynamically start & stop** the offloading threads
  - Using a user-level thread library (≈ a task-parallel library)

Implemention of our offloading method

- 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



- 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
- Solution: **Atomic operations + user-level threads**
  - Embed a bit whether the consumer is sleeping or not in the queue's counter, and if sleeping, awake the consumer using user-level threads
  - Faster than the kernel threading primitives (e.g. condition variables)

Evaluation of our communication library

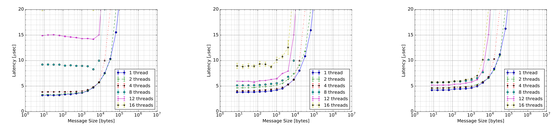
- Microbenchmark on these metrics:
  - Latency, overhead & message rate
  - Runs 2 processes (1 process/node) and one of them has benchmark threads repeating RDMA READ

CPU	Intel® Xeon® E5-2680 v2 2.80GHz, 2 sockets× 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")

- Used MassiveThreads 0.97 for user-level threading
  - Change to use parent-first scheduling (child-first is the default)
  - Run only 10 worker threads/node to avoid NUMA effects
- 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

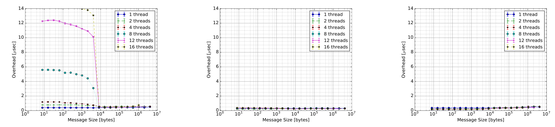
Microbenchmark results of our communication library

- **Latency** with 1 QP & CQ
  - Reference: 2.01 μsec in perftest benchmark
  - Offloading generally increases the latency



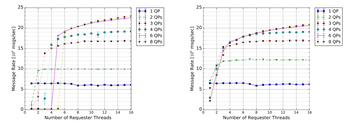
Direct injection | Static offloading | Dynamic offloading

- **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

- **Message rates** 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
    - The difference of 2 methods is how to wake up the consumer thread
    - Workers are out of resources in "Fork"
    - Additional synchronizations in "Condition variables"



Fork (parent-first) | Condition variables

Conclusions

- Runtime systems for global-view programming models
  - A **Distributed Shared Memory (DSM)** library
    - Transparent execution of shared-memory programs
  - A **communication library** for implementing the DSM
    - Software offloading for efficient fine-grained communications on multi-core architectures
- Future work
  - Analyze the bottlenecks of the DSM
  - Reduce the latency of software offloading

References

[1] `http://benchmark-subsetting.github.io/cNPB/`