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

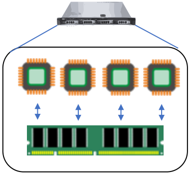
March 9, 2018 @ SIAM-PP

Summary of This Talk

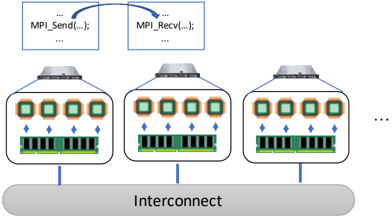
- Goal of our research:
  - Improve the productivity & performance of applications running on **distributed memory** machines
- We focus on **global-view** programming models:
  1. **Distributed Shared Memory (DSM)** as a unified memory model
  2. Global-view task scheduling as a unified execution model
- We are implementing 3 components:
  1. A software **DSM library** optimized for recent hardware
  2. A work-stealing scheduler over DSM
  3. A **communication library** for recent interconnects & multi-core processors

Background: Shared memory vs. Distributed memory

- **Shared memory**
  - All of the cores share the memory
  - Cores communicate implicitly though store/load instructions
  - High productivity, but low scalability

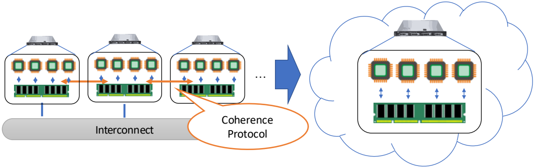


- **Distributed memory**
  - Each core (or node) has its own memory
  - Communications are explicit (e.g. MPI)
  - High scalability, but low productivity



Background: Distributed Shared Memory (DSM)

- **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], X10 [Charles et al. '05], Chapel [Chamberlain et al. '07], 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]

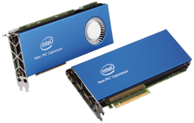
Latency

	DRAM	Internode
1990's	$\approx$ 100ns	$\approx 100\mu s$
2010's	$\approx$ 50ns	$\approx$ 500ns

Bandwidth

	DRAM	Internode
1990's	$\approx$ 2Gbps	$\approx$ 10Mbps
2010's	$\approx$ 250Gbps	$\approx$ 100Gbps

2. Relationship with **many-core** architectures
   - Shared memory is considered as a bottleneck of scalability
   - Techniques for DSM are revisited
     - e.g. Relaxed consistency, multiple-writer protocols

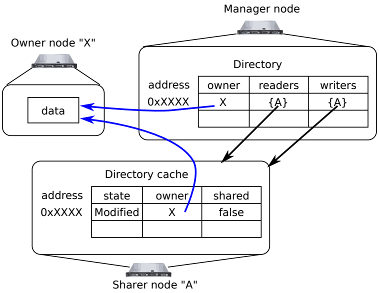


Intel Xeon Phi [1]

[1] https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

Design of First Prototype DSM

- Implemented a DSM prototype
- Borrowing the ideas from ArgoDSM [Kaxiras et al. '15]
  - Relaxed consistency (SC-for-DRF)
  - Page-based (vs. compiler-based)
  - Directory-based (vs. snoopy)
  - Multiple-Reader Multiple-Writer (vs. Single-Writer)
  - Home-based (vs. homeless)
  - Eager writeback (vs. Lazy)
- Added minor features (e.g. dynamic page allocation)

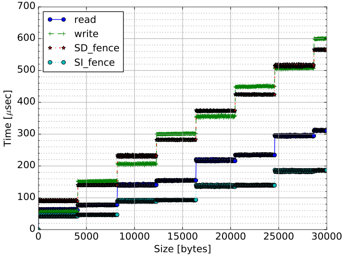


Directory structure of DSM

Evaluation: First Prototype DSM

- Microbenchmark of memory operations
  - Using 2 nodes of ReedBush-U, an InfiniBand cluster in our university
- Each memory operation consumes $\approx 100\mu\text{sec}$
  - cf. Round-trip RDMA latency $\approx 2\mu\text{sec}$
  - Room to improve the implementation quality
  - The write-back operation is the slowest
    - Due to packing & unpacking diffs for merging the dirty pages

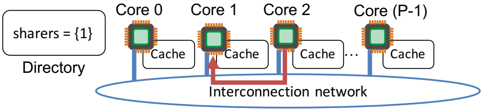
```
for (int k = 0; k < size; ++k)
    s += p[k]; // read
for (int k = 0; k < size; ++k)
    p[k] = x; // write
SD_fence(); // write back
SI_fence(); // invalidation
```



Horizontal axis: execution time, vertical axis: access size

Cache invalidation methods: Directory-based

- **Directory-based invalidations**
  - Tracking sharers in centralized directories
  - The standard method to implement large-scale shared memory
- Problems of directories:
  1. Storage cost to hold sharers
     - Less important in software-based shared memory
  2. **Communication traffic of small invalidation messages**
  3. Complex state management leads to system bugs



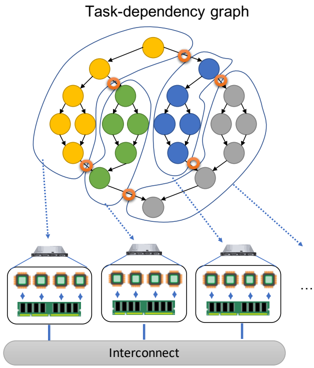
Cache invalidation methods: Directory-less

- "Directory-less" approaches:
  1. **Write notices**
     - Used in traditional DSM systems (e.g. TreadMarks)
     - Aggregate invalidation messages based on synchronization orders of relaxed consistency
  2. **(Logical-)timestamp-based coherence** [Yu et al. '15]
     - Invalidate cache blocks based on logical timestamps (= Lamport clocks)
- We are implementing a hybrid approach of write notice & timestamp-based approach [Yao et al. '16] in a software DSM
  - Not ready for evaluation now

Global-view Task Scheduling

- **Global-view task parallelism**
  - Dynamically schedule tasks beyond the nodes
  - Promising as a unified execution model for distributed systems
- Library-based implementation techniques:

1. User-level threading + work-stealing
   - Typical in shared-memory schedulers (.g. MassiveThreads [Nakashima et al. '14], QThreads [Wheeler et al. '08], Cilk [Blumofe et al. '95])
2. Iso-address
   - Globally allocate the same address range for each call stack
   - Used in distributed memory schedulers (e.g. Charm++ [Acun et al. '14])



Global-view Task Scheduling on DSM

- Typical distributed-memory schedulers are not fully transparent
  - Consider a simple program with a pointer dereference:

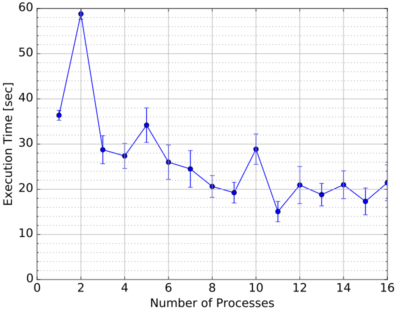
```
void f(void* p) {
    *(int*)p = 1;
    // Accessing a call stack of another thread
}
int g() {
    int x;
    thread_t t = thread_fork(&f, &x);
    thread_join(t);
    return x;
}
```

- Solution: **coherent call stacks**
  - Similar, but a compiler-based approach:
    (e.g. DAG Consistency [Blumofe et al. ’96])
  - We implemented a library-based method to manage call stacks in DSM
    - SIGSEGV handlers for call stacks are automatically disabled and avoid deadlocks

Evaluation: Global-view Task Scheduling on DSM

- Running a microbenchmark to measure the scheduler performance
  - Calculating `fib(30)` on the DSM & global-view work-stealing scheduler
  - 1 worker thread / node
- Did not scale well
  - The sequential performance was also unsatisfactory (264x worse than MassiveThreads)

```
void fib(int n, int* r) {
    if (n < 2) { *r = n; }
    else {
        int a, b;
        spawn fib(n-1, &a);
              fib(n-2, &b);
        sync;
        *r = a + b;
    }
}
```

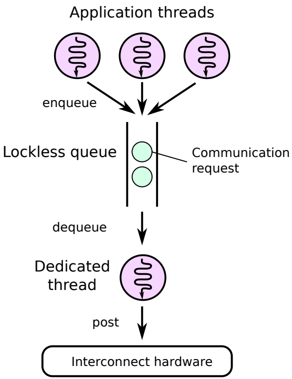


Communication library for DSM

- We 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
  - Traditional communication libraries are optimized for coarse-grained & single-threaded communications
- The rest of this talk briefly introduces:
  - Wataru Endo, Kenjiro Taura. “Parallelized Software Offloading of Low-Level Communication with User-Level Threads.” HPC Asia 2018.

Software Offloading

- We are focusing on **software offloading** [Vaidyanathan et al. ’15] to deal with small messages:
    - 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 MPI_THREAD_SERIALIZED
    - Only one thread handles actual MPI communication



Problems of Software Offloading

- 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

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

Evaluation

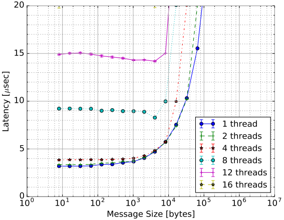
- Microbenchmark on these metrics:
    - Latency, overhead, and message rate
- Runs 2 processes (1 process/node)
    - One process has benchmark threads repeating RDMA READ
- MassiveThreads 0.97 as a ULT system
    - Change to use parent-first scheduling (child-first is the default)
    - Run only 10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

Evaluation Environment

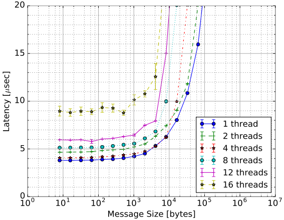
CPU	Intel® Xeon® E5-2680 v2 2.80GHz, 2 sockets× 10 cores/node
Memory	16GB/node
Interconnect	Mellanox® Connect-IB® dual port InfiniBand FDR 2-port (only 1 port is used)
Driver	Mellanox® OFED 2.4-1.0.4
OS	Red Hat® Enterprise Linux® Server release 6.5 (Santiago)
Compiler	GCC 4.4.7 (with the option “-O3”)

Evaluation Results: Latency with 1 QP & CQ

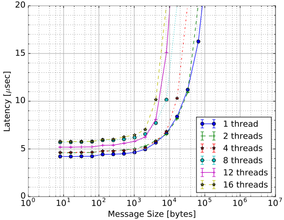
- Reference: 2.01 μsec in perftest benchmark
- 3.197 μsec in Direct injection
  - Overhead of separating polling threads
- 3.804μsec in Static offloading
  - Overhead of sending a request to an executor thread
- 4.21μsec in Dynamic offloading
  - Overhead of waking up an executor & completer thread



Direct injection



Static offloading

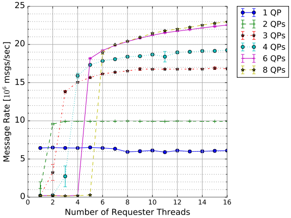


Dynamic offloading

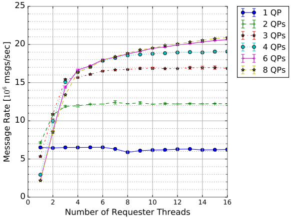
Horizontal axis: message size. Vertical axis: round-trip latency.
Each line represents # of requester threads.

Evaluation Results: 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.

Conclusions

- Runtime systems for global-view programming models
  - A **Distributed Shared Memory (DSM)** library
  - A **global-view task scheduler** based on the DSM
    - Transparent execution of shared-memory task-parallel programs
  - A **communication library** for implementing the DSM
    - Software offloading for efficient fine-grained communications on multi-core architectures
- Future work
  - Implement a directory-less DSM system
  - Evaluate on real applications