Uni-Address Threads: Scalable Thread Management for RDMA-based Work Stealing

Shigeki Akiyama, Kenjiro Taura
The University of Tokyo
June 17, 2015  HPDC’15
Lightweight Threads

- Lightweight threads enable us to create a large number of threads
  - We can express logical concurrency as a thread
  - Runtime system performs dynamic load balancing
  - e.g. MassiveThreads, Qthreads, Nanos++

```c
long fib(long n) {
    if (n < 2) {
        return n;
    } else {
        thread_t t0 = thread_create(fib, n - 1);
        thread_t t1 = thread_create(fib, n - 2);
        long r0 = thread_join(t0);
        long r1 = thread_join(t1);
        return r0 + r1;
    }
}
```
Work Stealing

• A promising approach to dynamic load balancing
  - Each processor has a task queue
  - Idle processor steals tasks from another processor
Introducing inter-node work stealing to lightweight multithreading is challenging:

- It needs to migrate threads among nodes
- An existing thread migration scheme (iso-address) is not scalable:
  - Each node requires \( O(P) \) virtual memory for thread stacks
  - Thread migration cannot utilize Remote Direct Memory Access (RDMA) features

Important for scalability of work stealing in large-scale distributed memory systems [Dinan,09]
Goal

- Lightweight multithread library supporting scalable inter-node work stealing
  - Solve scalability issues in existing thread migration scheme
    - Significantly reduce virtual memory usage
    - Enable RDMA-based thread migration
Contributions

- Propose a new thread migration scheme, **uni-address**
  - requires **only O(1) virtual memory per node** for thread stacks

- Implement a lightweight multithread library based on uni-address scheme
  - Scalable work stealing by RDMA features

- Demonstrate its efficiency and scalability up to 4000 cores on Fujitsu FX10 system
Related Work:
Global Load Balancing Frameworks

• Classify them with implementation strategies
  - Bag-of-Tasks
  - Fork-join with tied tasks
  - Fork-join with untied tasks
Related Work: Global Load Balancing Frameworks

- Bag-of-Tasks
  - Tasks cannot synchronize with other tasks
  - Task = a function pointer + arguments
    - Easy to implement task migration
  - cf. Scioto [Dinan08], X10-GLB [Zhang08]
Related Work: Global Load Balancing Frameworks

• Fork-join with tied tasks
  - Support fork-join synchronization between tasks
  - Task = a function pointer + arguments
    ‣ Easy to implement task migration
  - Tasks are tied: task already started cannot migrate
    ‣ Low flexibility of task scheduling:
      e.g. lower load balancing efficiency
  - cf. Satin [Neuwpoort01], HotSLAW [Min11]
Related Work: Global Load Balancing Frameworks

• Fork-join with untied tasks
  - Support fork-join synchronization and task migration at any program point
  - Compiler-based
    ‣ cf. Distributed Cilk [Blumofe96], Tascell [Hiraishi09]
  - Library-based
    ‣ Task = thread (which have a call stack)
    ‣ Difficulty in migration of a call stack beyond node boundary
    ‣ iso-address [Antoniu99] and our work solved it
## Related Work: Global Load Balancing Frameworks

<table>
<thead>
<tr>
<th></th>
<th>Inter-task synchronization</th>
<th>Untied tasks</th>
<th>library/ compiler</th>
<th>Demonstrated Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scioto [Dinan09]</td>
<td>×</td>
<td>×</td>
<td>library</td>
<td>8192</td>
</tr>
<tr>
<td>X10-GLB [Zhang13]</td>
<td>×</td>
<td>×</td>
<td>library</td>
<td>16384</td>
</tr>
<tr>
<td>Satin [Neuwpoort01]</td>
<td>fork-join</td>
<td>×</td>
<td>compiler</td>
<td>256</td>
</tr>
<tr>
<td>HotSLAW [Min11]</td>
<td>fork-join</td>
<td>×</td>
<td>library</td>
<td>256</td>
</tr>
<tr>
<td>Distributed Cilk [Blumofe96]</td>
<td>fork-join</td>
<td>○</td>
<td>compiler</td>
<td>16</td>
</tr>
<tr>
<td>Tascell [Hiraishi09]</td>
<td>fork-join</td>
<td>○</td>
<td>compiler</td>
<td>128</td>
</tr>
<tr>
<td><strong>Proposed method</strong></td>
<td>fork-join</td>
<td>○</td>
<td>library</td>
<td>4096</td>
</tr>
</tbody>
</table>

The proposed method supports all of flexible task model, library-based implementation, and scalability.

---

[Dinan09]...

[Zhang13]...

[Neuwpoort01]...

[Min11]...

[Blumofe96]...

[Hiraishi09]...
Thread Migration

• Move a thread among nodes
  - A thread contains a call stack
  - Stack transfer may **invalidate intra-stack pointers**

![Diagram showing thread stack migration and invalid pointer](image)

**Naive stack transfer**

**We must maintain an invariant:**
the address of a call stack is the same around thread migration
Iso-Address: Existing Thread Migration Scheme [Antoniu,99]

- Put a stack on the same address around migration

- Allocate an unique address for a call stack to ensure the address is not used in the receiving node
  - requires $O(P)$ virtual memory per node
Scalability Issue 1

• **A large amount of virtual memory**
  - e.g.

  Stack size of a thread \(\approx 16\text{KB} = 2^{14}\)
  
  Recursion depth of thread creation \(\approx 8192 = 2^{13}\)
  
  Available cores \(\approx 4\text{ million} = 2^{22}\)

  In total: \(2^{14+13+22} = 2^{49} > 2^{48}\)

  exceeds x86-64 virtual memory limit
Scalability Issue 2

- Unable to implement RDMA-based thread migration, important for scalable load balancing [Dinan, 09]
  - Because:
    - RDMA-capable memory must be pinned down to physical memory
    - Virtual memory usage of iso-address is too large to fit into physical memory
Basic Idea of Uni-Address Scheme

• Iso-address
  - A stack must be copied to the same virtual address in the receiving node upon migration
  - i.e. a stack **ALWAYS** occupies the same address

• Uni-address
  - Key observation: it suffices to occupy the same address **WHEN THE THREAD IS RUNNING**
  - Reduce virtual memory usage by placing not-running threads into arbitrary addresses
Basic Uni-Address Scheme

- Ensure that all threads shares the same address region, *uni-address region*
  - Place a running thread on the uni-address region
  - Not-running threads are evicted to RDMA-capable region

**Context switch in uni-address scheme**

![Diagram showing context switch in uni-address scheme]
Basic Uni-Address Scheme

- Ensure that all threads shares the same address region, *uni-address region*
  - Place a running thread on the uni-address region
  - Not-running threads are evicted to RDMA-capable region

Context switch in uni-address scheme

1. Save execution context and *evict stack contents to RDMA region*
Basic Uni-Address Scheme

• Ensure that all threads shares the same address region, *uni-address region*
  - Place a running thread on the uni-address region
  - Not-running threads are evicted to RDMA-capable region

**Context switch in uni-address scheme**

1. Save execution context and **evict stack contents to RDMA region**

```
<table>
<thead>
<tr>
<th>RDMA region</th>
</tr>
</thead>
<tbody>
<tr>
<td>(arbitrary address)</td>
</tr>
<tr>
<td>Thread A (running)</td>
</tr>
<tr>
<td>Thread C (not-running)</td>
</tr>
<tr>
<td>Thread B (not-running)</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>Uni-address region</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xXXXX0000</td>
</tr>
<tr>
<td>0xYYYY0000</td>
</tr>
<tr>
<td>← sp</td>
</tr>
</tbody>
</table>
```

Thread A (running)
Basic Uni-Address Scheme

- Ensure that all threads shares the same address region, *uni-address region*
  - Place a running thread on the uni-address region
  - Not-running threads are evicted to RDMA-capable region

**Context switch in uni-address scheme**

1. Save execution context and **evict stack contents to RDMA region**

2. **Load stack contents to uni-address region** and resume execution context
Basic Uni-Address Scheme

- Ensure that all threads shares the same address region, *uni-address region*
  - Place a running thread on the uni-address region
  - Not-running threads are evicted to RDMA-capable region

**Context switch in uni-address scheme**

1. Save execution context and **evict stack contents to RDMA region**

2. **Load stack contents to uni-address region** and resume execution context
Optimized Uni-Address Scheme

• Problem with basic uni-address scheme
  - Context switch **incurs two stack copies**: thread operations become heavyweight

• How to reduce the stack copies?
  - **Put two or more threads in uni-address region** to reduce thread eviction
  - **Focus on thread creation/exit operations** because of

    
    (# of thread creation) >> (# of load balancing ops)
Thread Scheduling in Optimized Scheme

- Child-first work stealing scheduler (cf. [Mohr,91], [Frigo,98])
  - Execute a thread creation as if it is a function call
  - Can allocate child’s stack right above the parent stack

Optimized scheme can create threads without stack copy
Thread Scheduling in Optimized Scheme

- Child-first work stealing scheduler (cf. [Mohr,91], [Frigo,98])
  - Fork-join synchronization suspends a thread when the child thread is on another processor
  - A thread is evicted only when work stealing occurs

![Diagram showing thread scheduling and eviction](image-url)
Experimental Evaluation

• We implemented a lightweight multithread library based on uni-address scheme
  - Implemented inter-node work stealing with RDMA operations

• Evaluate
  - Threading overhead
  - Work stealing time
  - Load balancing scalability with task-parallel benchmarks
Experimental Setup

• Environments
  - Fujitsu FX10 system up to about 4000 cores
    ‣ Simulate remote atomic operations with one assistant core per node
  - a Xeon E5-2660 2.2GHz server

• Load balancing benchmarks:
  - Binary Task Creation
  - Unbalanced Tree Search
  - NQueens solver
Thread Creation Overhead

<table>
<thead>
<tr>
<th></th>
<th>SPARC64IXfx</th>
<th>Xeon E5-2660</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uni-address threads</td>
<td>413 cycles</td>
<td>100 cycles</td>
</tr>
<tr>
<td>MassiveThreads</td>
<td>658 cycles</td>
<td>110 cycles</td>
</tr>
</tbody>
</table>

- Comparable to MassiveThreads, an existing lightweight multithread library
  - thanks to optimized uni-address scheme
Breakdown of Work Stealing Time

- **42K cycles** in total
- Overhead originating from uni-address scheme is **3.5K cycles** (**7% of total work stealing time**)
Load Balancing Scalability (~3840 cores)

### Table 3: Operations consisting of work stealing.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty check</td>
<td>A operation to check whether a remote task queue is empty or not. It consists of an RDMA READ operation.</td>
</tr>
<tr>
<td>lock</td>
<td>A lock operation for a remote task queue. It consists of a remote fetch-and-add operation.</td>
</tr>
<tr>
<td>steal</td>
<td>An operation to steal an entry from a remote task queue. It consists of two RDMA READ and an RDMA WRITE operations.</td>
</tr>
<tr>
<td>suspend</td>
<td>An operation to suspend a running thread.</td>
</tr>
<tr>
<td>stack transfer</td>
<td>An operation to transfer stack frames. It consists of an RDMA READ operation.</td>
</tr>
<tr>
<td>unlock</td>
<td>A unlock operation for a remote task queue. It consists of an RDMA WRITE operation.</td>
</tr>
<tr>
<td>resume</td>
<td>An operation to resume a stolen thread.</td>
</tr>
</tbody>
</table>

### Table 4: The number of generated tasks or nodes in three benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
<th>Total tasks or nodes</th>
<th>Time (sec)</th>
<th>Stack usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Task Creation (iter=1)</td>
<td>depth = 38</td>
<td>8 billion tasks</td>
<td>55.67</td>
<td>43,568</td>
</tr>
<tr>
<td></td>
<td>depth = 39</td>
<td>9 billion tasks</td>
<td>33.37</td>
<td>44,688</td>
</tr>
<tr>
<td></td>
<td>depth = 19</td>
<td>9 billion tasks</td>
<td>32.96</td>
<td>22,288</td>
</tr>
<tr>
<td></td>
<td>depth = 20</td>
<td>10 billion tasks</td>
<td>88.14</td>
<td>23,408</td>
</tr>
<tr>
<td>Unbalanced Tree Search</td>
<td>depth = 17</td>
<td>11 billion nodes</td>
<td>71.62</td>
<td>139,536</td>
</tr>
<tr>
<td></td>
<td>depth = 18</td>
<td>43 billion nodes</td>
<td>282.2</td>
<td>147,392</td>
</tr>
<tr>
<td>NQueens</td>
<td>N = 17</td>
<td>8 billion nodes</td>
<td>47.60</td>
<td>74,272</td>
</tr>
<tr>
<td></td>
<td>N = 18</td>
<td>59 billion nodes</td>
<td>317.8</td>
<td>79,120</td>
</tr>
</tbody>
</table>

**Figure 11: Parallel performance in three benchmarks.**

- **Efficiency:** 98%
- **Efficiency:** 84%
- **Efficiency:** 99%
- **Efficiency:** 95%

All the benchmarks worked with 144KB uni-address region.
Summary

• Uni-address: A scalable thread migration scheme
  - Requires only $O(1)$ virtual memory per node
  - Enables RDMA-based work stealing

• Demonstrated its performance with FX10 system
  - Comparable threading overhead to an existing library
  - Load balancing scalability up to 4000 cores