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

Shigeki Akiyama, Kenjiro Taura

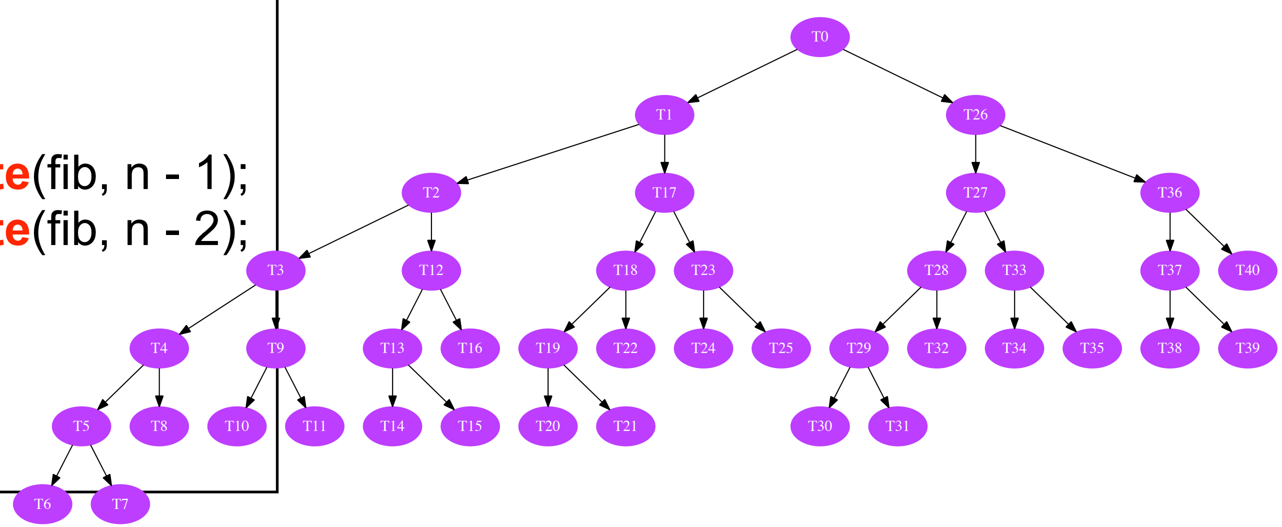
The University of Tokyo

June 17, 2015 HPDC'15

Lightweight Threads

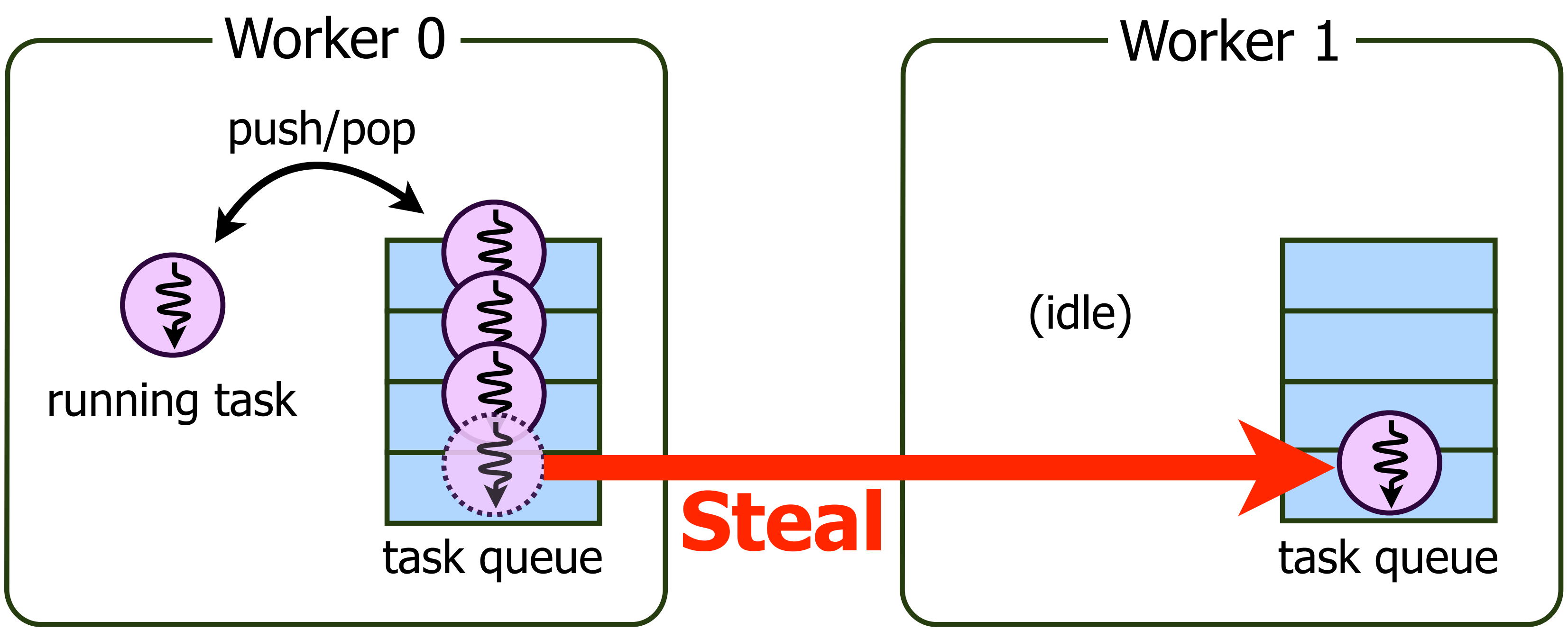
- Lightweight threads enable us to create a large number of threads
  - We can express logical concurrency as a thread
  - Runtime system performs dynamic load balancing
  - e.g. MassiveThreads, Qthreads, Nanos++

```
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



Inter-Node Work Stealing with Lightweight Threads

- 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 <u>fork-join synchronization</u> 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 <u>fork-join synchronization</u> and <u>task migration at any program point</u>
  - Compiler-based
    - cf. Distributed Cilk [Blumofe96], Tascell [Hiraishi09]
  - Library-based
    - <u>Task = thread</u> (which have a call stack)
    - **Difficulty in migration of a call stack beyond node boundary**
    - iso-address [Antoniu99] and our work solved it

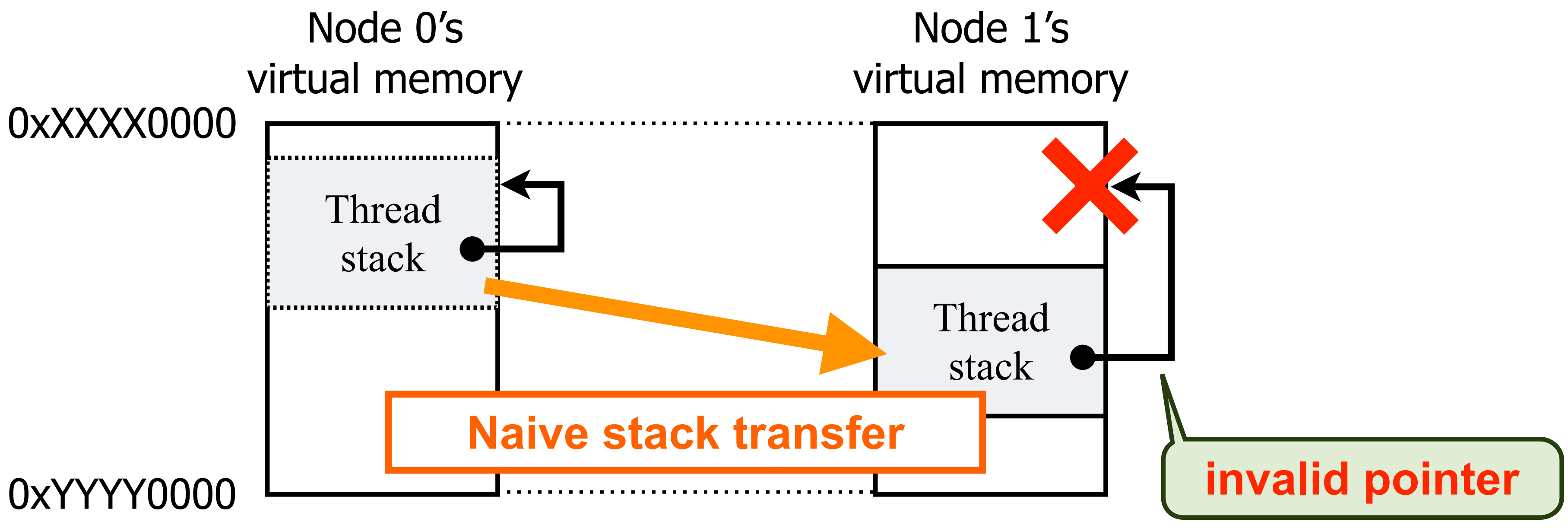
Related Work: Global Load Balancing Frameworks

	Inter-task synchronization	Untied tasks	library/ compiler	Demonstrated Scalablity
Scioto [Dinan09]	×	×	library	8192
X10-GLB [Zhang13]	×	×	library	16384
Satin [Neuwpoort01]	fork-join	×	compiler	256
HotSLAW [Min11]	fork-join	×	library	256
Distributed Cilk [Blumofe96]	fork-join	○	compiler	16
Tascell [Hiraishi09]	fork-join	○	compiler	128
Proposed method	fork-join	○	library	4096

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

Thread Migration

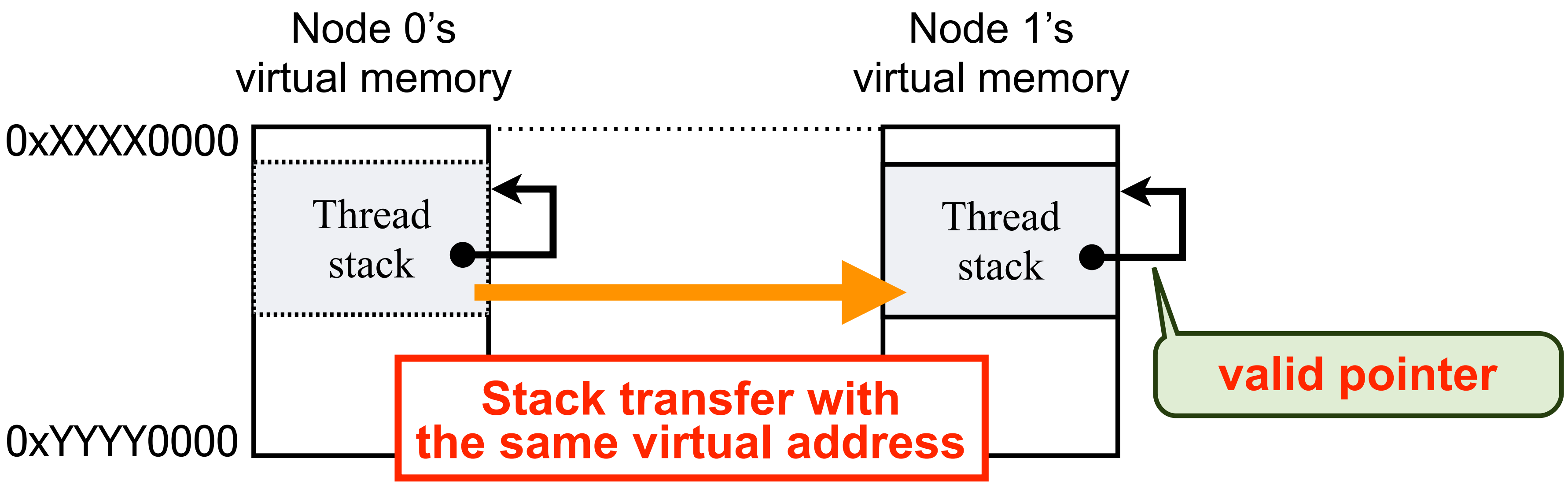
- Move a thread among nodes
  - A thread contains a call stack
  - Stack transfer may **invalidate intra-stack pointers**



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 <u>an unique address for a call stack</u>
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.

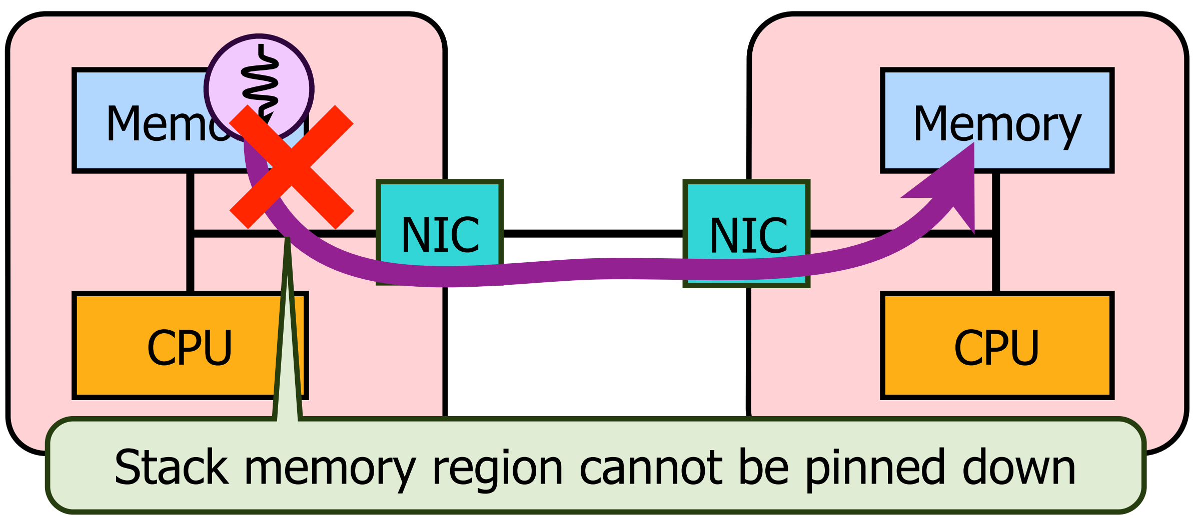
Stack size of a thread	≈ 16KB = $\mathbf{2^{14}}$
Recursion depth of thread creation	≈ 8192 = $\mathbf{2^{13}}$ (cf. Unbalanced Tree Search)
Available cores	≈ 4 million = $\mathbf{2^{22}}$ (cf. Tianhe-2)

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

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

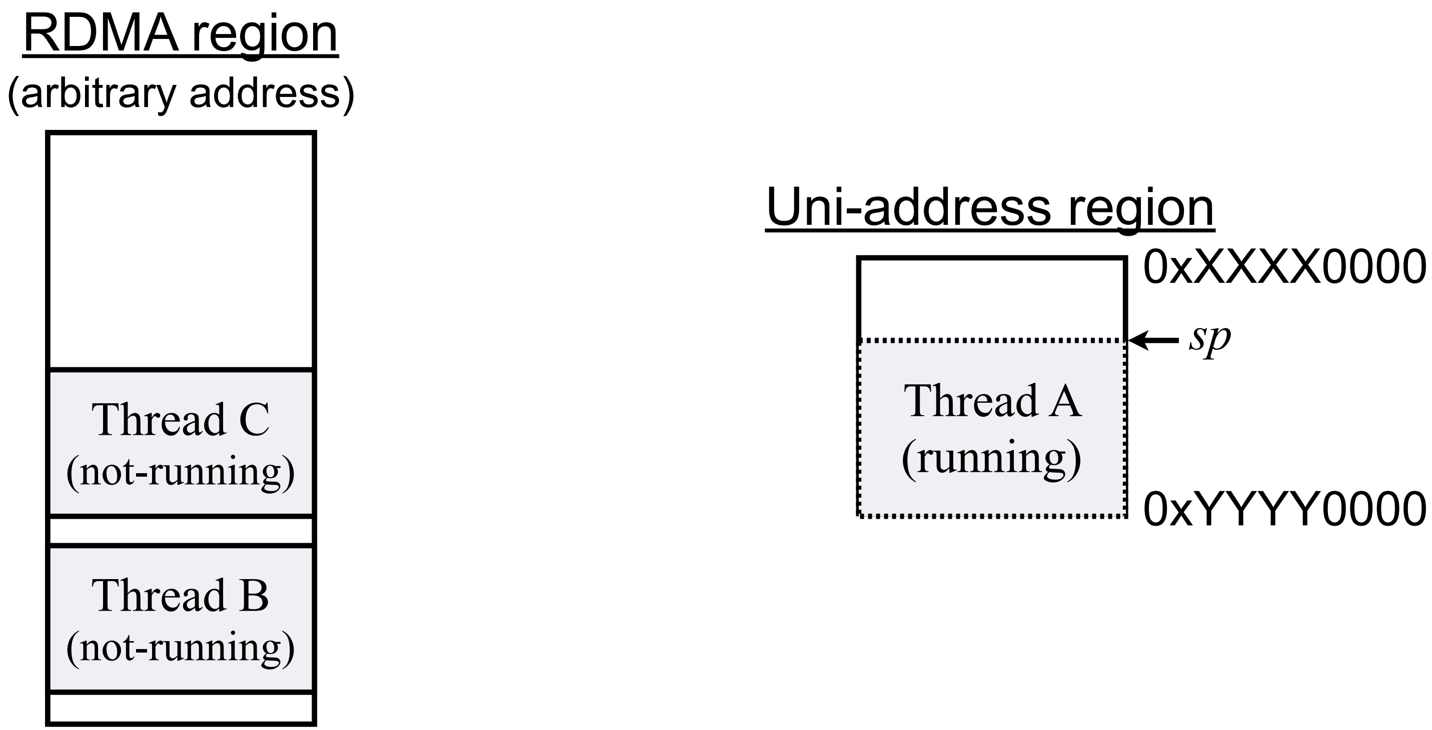
- Iso-address
  - A stack must be copied to the same virtual address in the receiving node upon migration
  - i.e. a stack **ALWAYS** occupies the same address

- **Uni-address**
  - Key observation: it suffices to occupy the same address **WHEN THE THREAD IS RUNNING**
  - Reduce virtual memory usage by <u>placing not-running threads into arbitrary addresses</u>

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

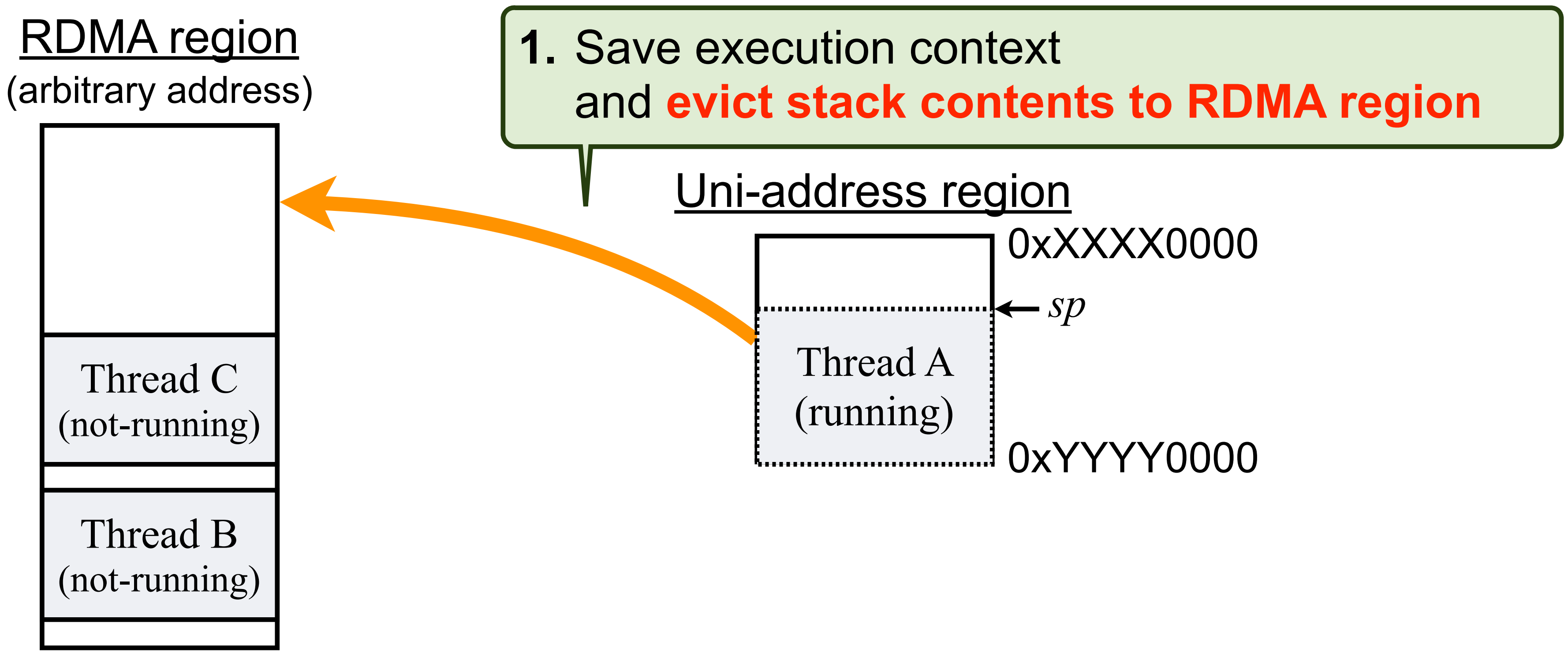
Context switch in uni-address scheme



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

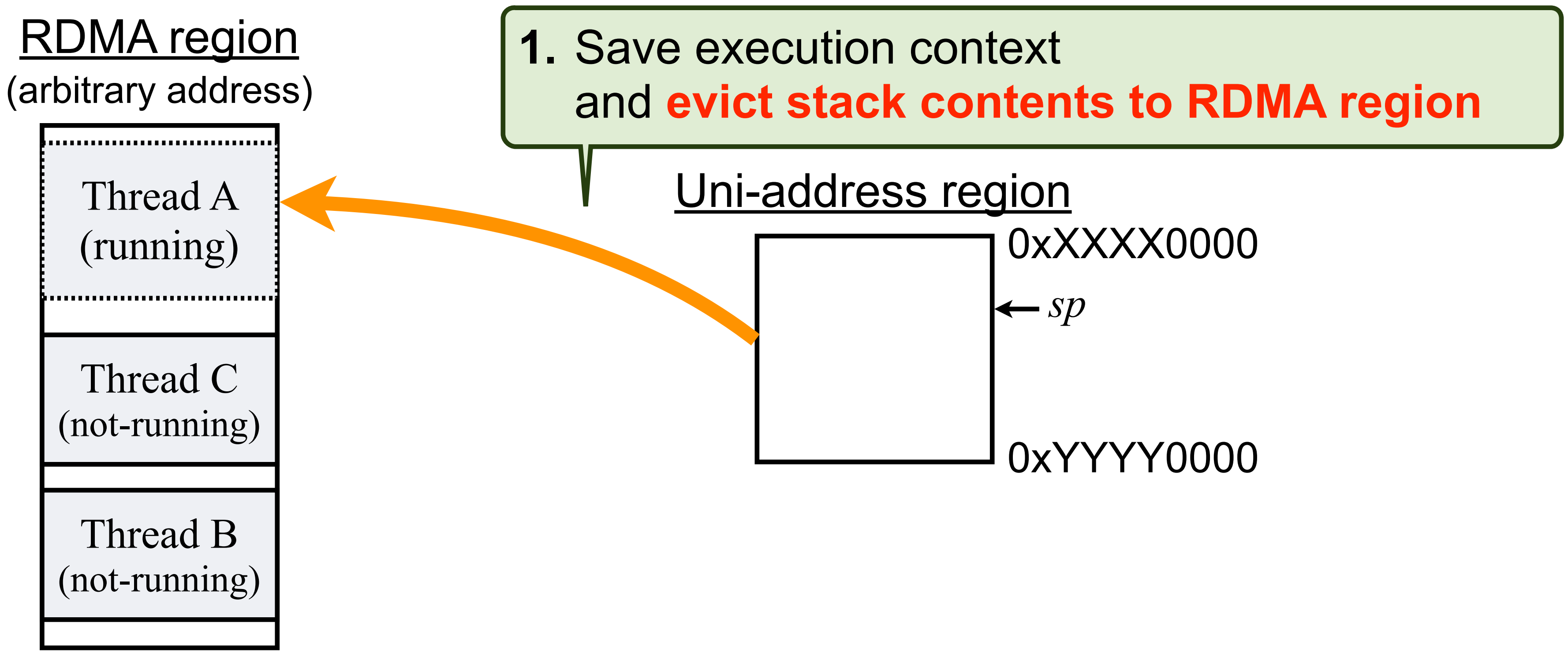
Context switch in uni-address scheme



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

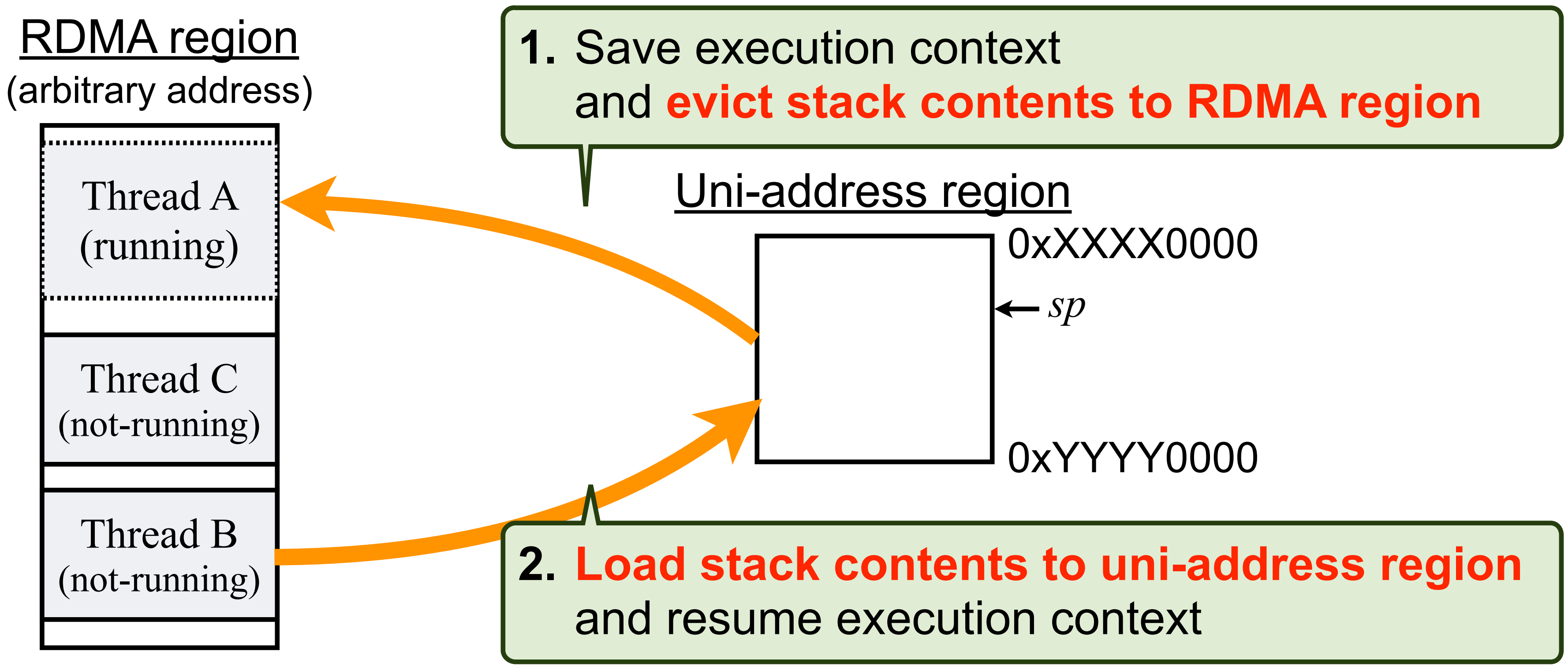
Context switch in uni-address scheme



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

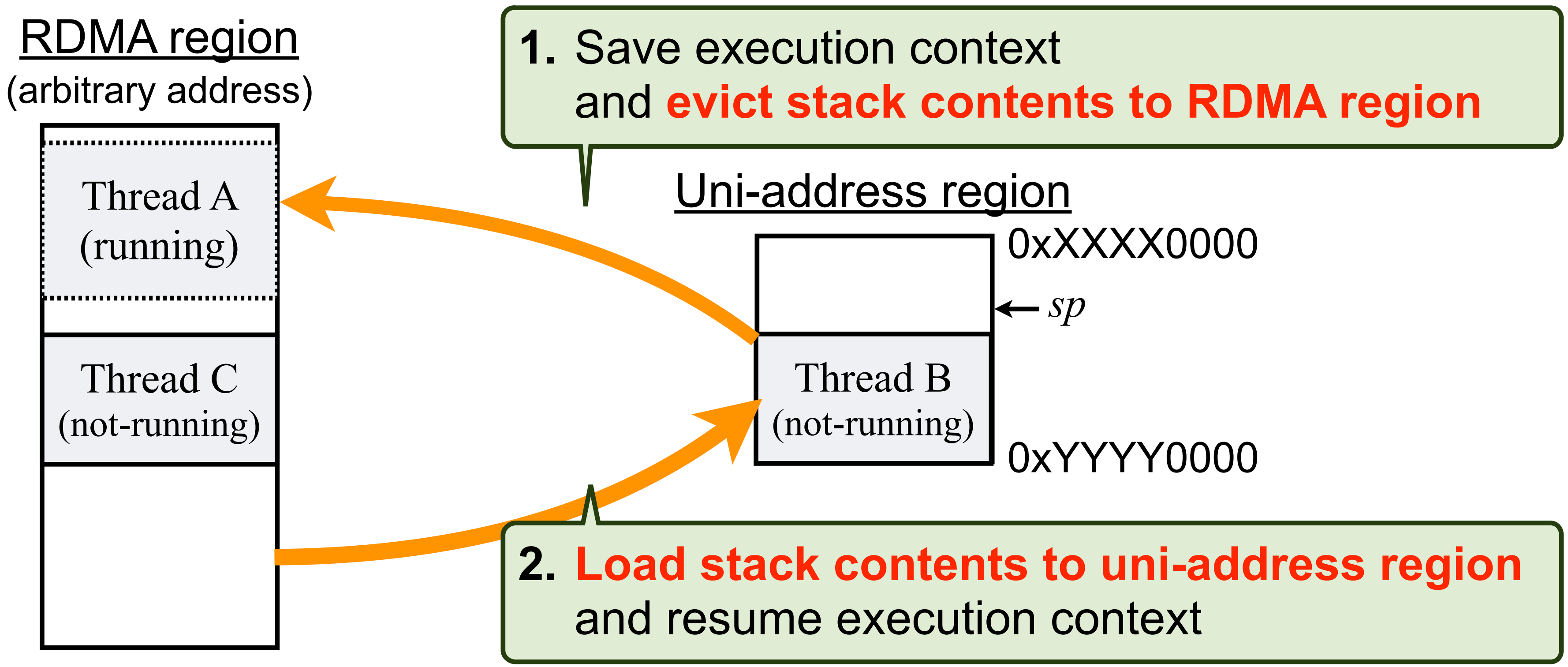
Context switch in uni-address scheme



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



Optimized Uni-Address Scheme

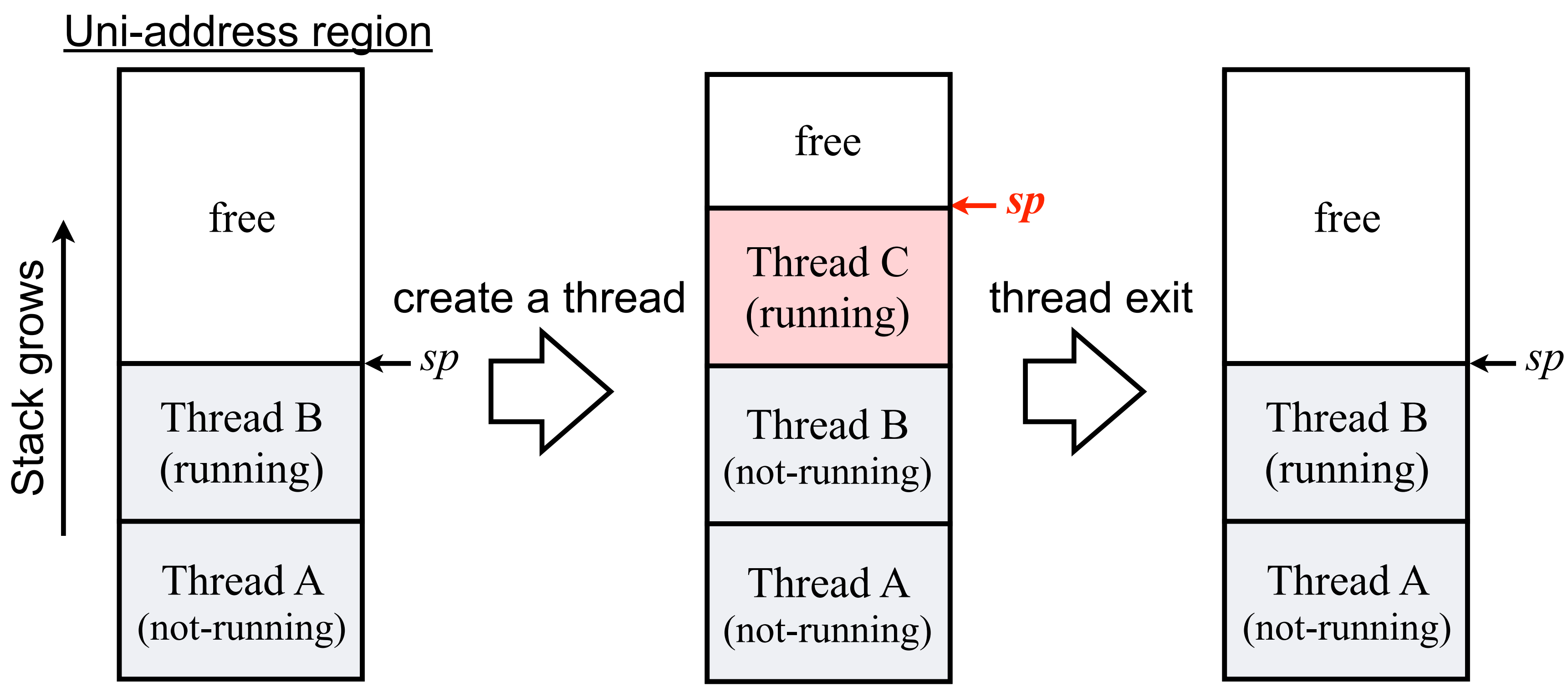
- Problem with basic uni-address scheme
  - Context switch **incurs two stack copies**: thread operations become heavyweight

- 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

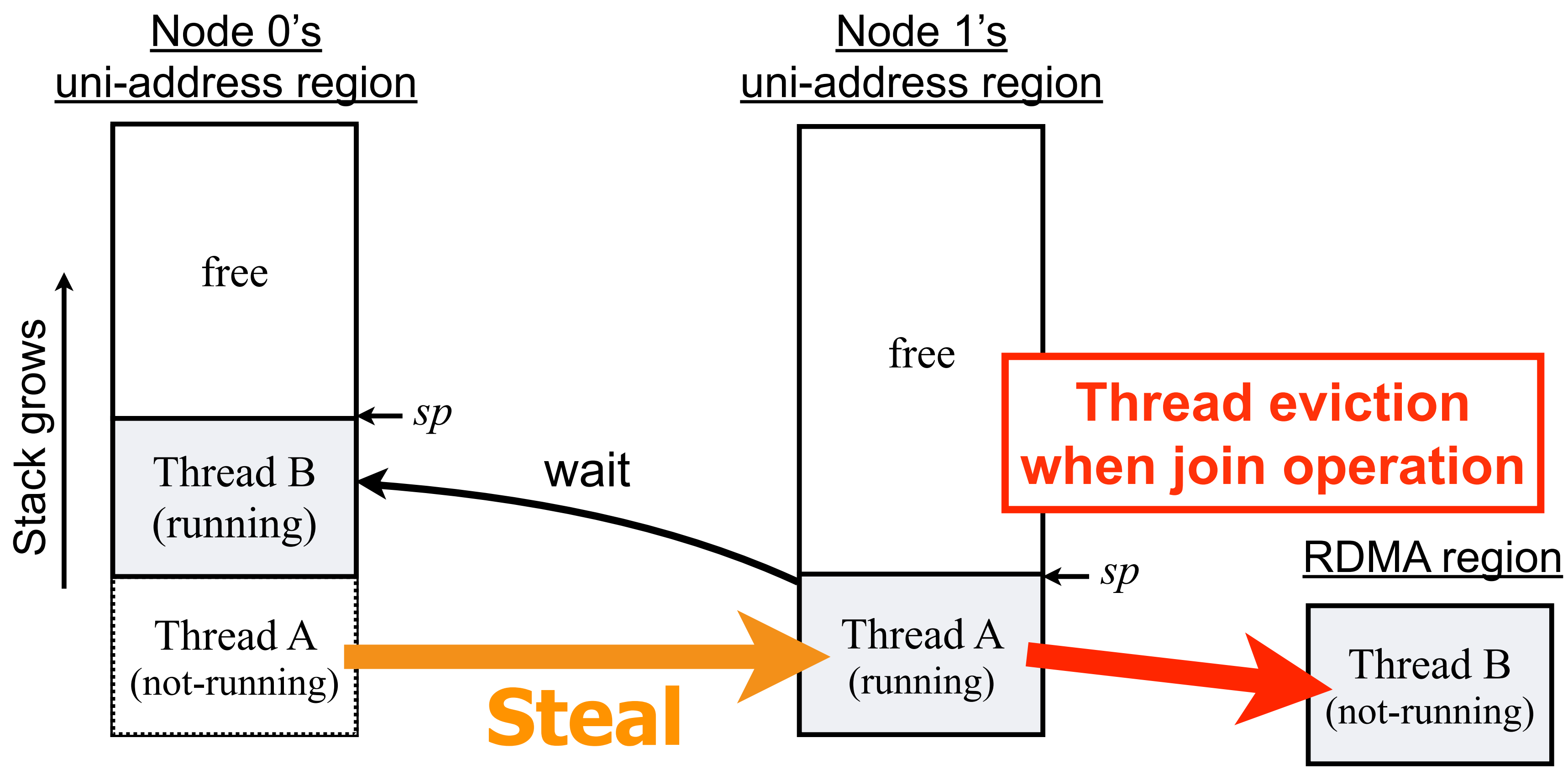
- Child-first work stealing scheduler (cf. [Mohr,91], [Frigo,98])
  - Execute a thread creation as if it is a function call
  - <u>Can allocate child's stack right above the parent stack</u>



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**



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

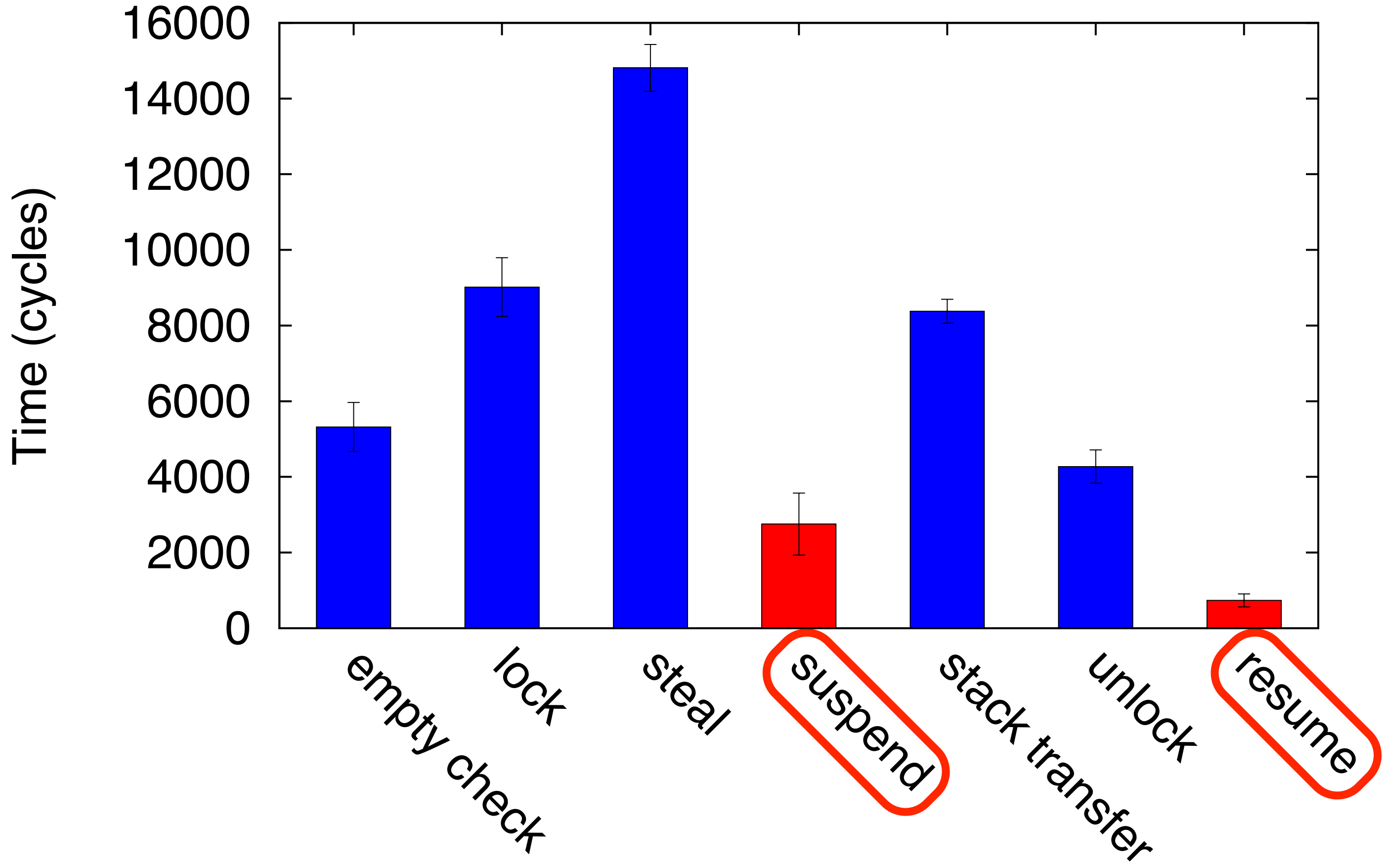
- Load balancing benchmarks:
  - Binary Task Creation
  - Unbalanced Tree Search
  - NQueens solver

Thread Creation Overhead

	SPARC64IXfx	Xeon E5-2660
Uni-address threads	413 cycles	100 cycles
MassiveThreads	658 cycles	110 cycles

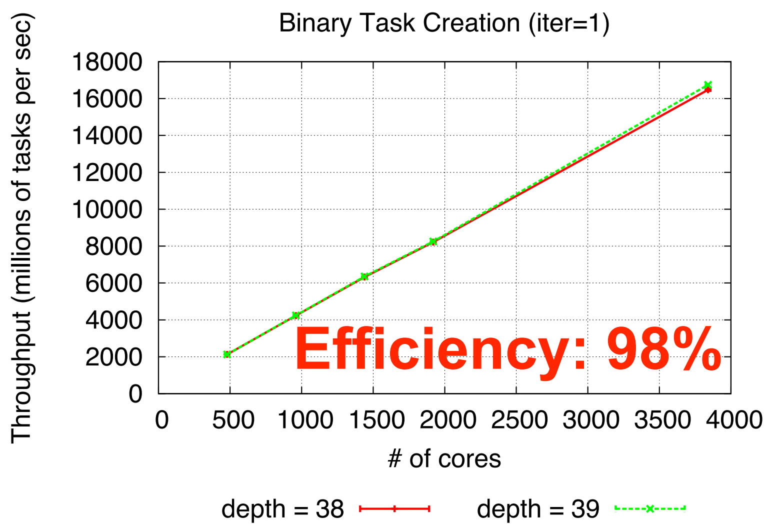
- Comparable to MassiveThreads, an existing lightweight multithread library
  - thanks to optimized uni-address scheme

Breakdown of Work Stealing Time

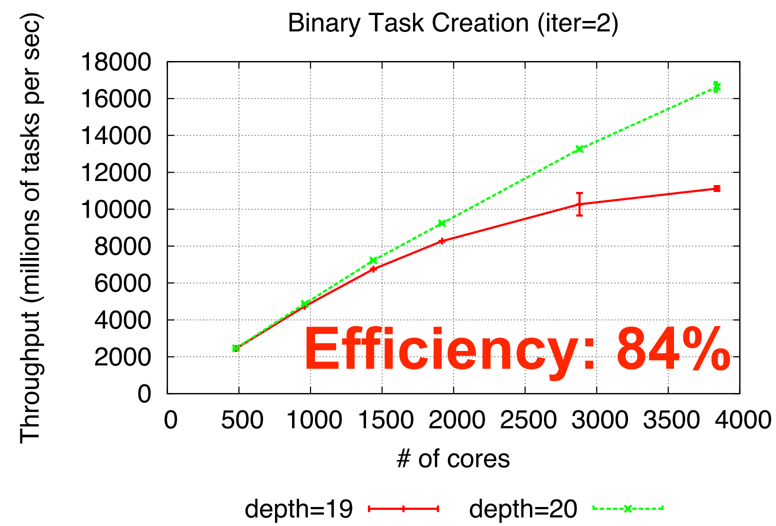


- 42K cycles in total
- Overhead originating from uni-address scheme is 3.5K cycles (**7%** of total work stealing time)

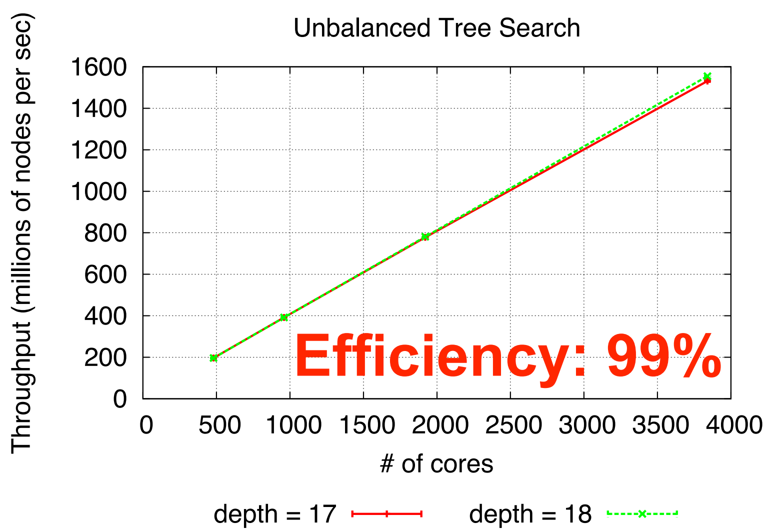
Load Balancing Scalability (~3840 cores)



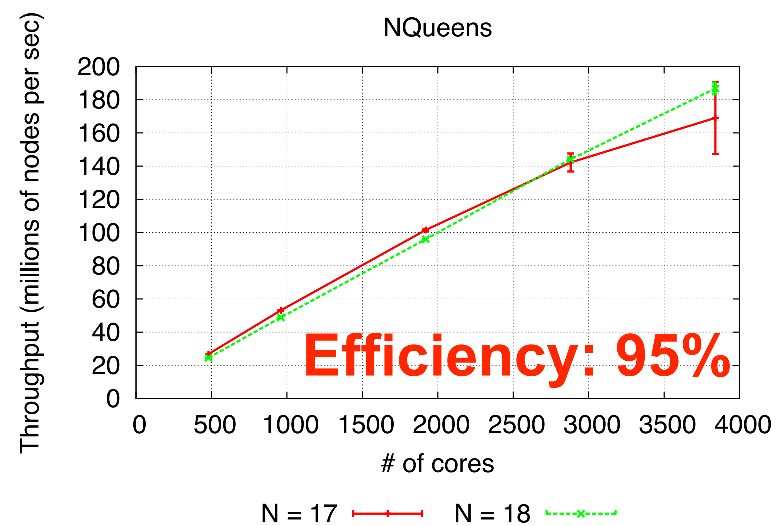
(a) Binary Task Creation (iter=1)



(b) Binary Task Creation (iter=2)



(c) Unbalanced Tree Search



(d) NQueens

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