1 Introduction

2 Preliminaries on Infiniband & RDMA

3 Part 1: Storing State Spaces

4 Part 2: Load-Balancing

5 Part 3: Distributed BDD Operations

6 Experimental Evaluation

7 Conclusion
What is Reachability Analysis?

- **Reachability Problem:** Given a graph $G = (V, E)$, initial states $I \subseteq V$ and goal states $F \subseteq V$, check if $F$ is reachable from $I$ via edges in $E$.

- **In Model Checking:** Allows verification of most temporal safety properties ("something bad will never happen")

- **Examples:**
  - Finding solutions in games (e.g. Chess, Sokoban, etc.)
  - Asserting mutual exclusion in parallel software
  - Asserting safety in traffic lights
  - etc.
State Space Explosions

- **Reachability Problem:** Given a graph \( G = (V, E) \), initial states \( I \subseteq V \) and goal states \( F \subseteq V \), check if \( F \) is reachable from \( I \) via edges in \( E \).

- \( G \) is often *implicitly* described *(with a transition relation)*
- Set of reachable states is often determined *on-the-fly*
- Therefore, the size of \( G \) is often initially unknown

- **State Space Explosions:** occur when \( G \) does not longer fit into the available memory
  - Often happens in practice \( \implies \) major limitation
  - State space Chess: \( \sim 10^{43} \)
  - Stars in universe: \( \sim 10^{23} \)
Fighting State Space Explosions

- **Reduction Techniques:**
  - Partial Order Reduction *(exploit commutative transitions)*
  - Bisimulation Minimization *(merge "similar" states)*

- **Compression Techniques:**
  - Decision Diagrams *(e.g. BDDs, MDDs, LDDs, ZDDs)*
  - SAT-based approaches *(e.g. IC3)*

- **Adding Hardware Resources:**
  - More memory $\rightarrow$ *larger* state spaces supported
  - More processors $\rightarrow$ *faster* reachability analysis
Parallel Symbolic Reachability

**Symbolic Reachability:**
- Represent the state space as a BDD
- Represent initial states and the transition relation as BDDs
- Perform reachability via BDD operations

**Parallel Reachability:** Using a many-core cluster with a large amount of memory to perform reachability
- Sylvan reaches speedups up to 38 with 48 cores

**Disadvantages:**
- Upgrading is *expensive*
- Upgrading is *limited*
Distributed Reachability: Using a network of workstations, connected via a high-performance network.

Compared to a Many-core Cluster:
- Cheaper scalability
- Unlimited scalability

Challenges:
- Only small amounts of computation per memory access
- Many remote memory accesses required
- Network latency easily becomes a bottleneck

Achievements: Very large state spaces are supported, but no speedups are obtained...
Zhao et al (2009): Most important design considerations for improvements are:
- Data-distribution
- Load-balancing maintenance
- Reducing communication overhead
- Exploiting data-locality *(suggested by Chung et al)*

Contribution: Employing *modern* techniques to implement these design considerations.
Motivation

- **Reducing Communicational Overhead**: Infiniband and RDMA

- **Load-balancing**: Work stealing (*due to the success of Sylvan and Lace*)

- **Exploiting data-locality**: *Hierarchical* work stealing

- **Data Distribution**: RDMA-based distributed hash table
Main Question: How efficient can RDMA-based distributed implementations of BDD operations scale along all processing units and available memory connected via a high-performance network?

Subquestions:

1. How can the storage and retrieval of data efficiently be managed to minimize their latencies?
2. How can the total computational work be divided and distributed to maximize scalability along processors over a high-performance network?
3. How can the idle-times of processes be minimized while performing network communication?
Project: We split the project into three parts:

1. Storing states (Distributed hash table)
2. Load-balancing mechanisms (Hierarchical work stealing)
3. Distributed BDD operations

All parts have separately been designed, implemented, and experimentally evaluated
# Table of Contents

1. Introduction
2. Preliminaries on Infiniband & RDMA
3. Part 1: Storing State Spaces
4. Part 2: Load-Balancing
5. Part 3: Distributed BDD Operations
6. Experimental Evaluation
7. Conclusion
Infiniband: Specialized hardware used to construct high-performance networks

Advantages:

1. Comparable in price to standard Ethernet hardware
2. Supports up to 100 GB/s
3. NICs can directly access main-memory via PCI-E bus
4. End-to-end latencies of 1 µs have been measured (according to the IB website)
5. Supports RDMA

UTwente: 10 Dell M610 machines, connected via a 20 GB/s Infiniband network
Remote Direct Memory Access

- **RDMA**: Directly access memory of a remote machine, without invoking its CPU
  - one-sided vs. two-sided RDMA

- **Advantages:**
  1. Zero-copy
  2. Kernel bypassing
  3. CPU efficiency

- **Roundtrip Latency**: Within 3\(\mu s\) in Infiniband hardware, compared to 60\(\mu s\) with TCP on Ethernet hardware
Kernel Bypassing

(VMWorld 2013 - How Latency Destroys Performance... And What to Do About It)
Parallel Programming Models

Shared Memory
- Single addr. space
- e.g. NUMA, SMP

Distributed Memory
- Only local memories
- Communication via Message Passing
Partitioned Global Address Space

**PGAS**
- Shared + Distributed
- Data locality exploited

**Hybrid PGAS**
- PGAS + message passing

Distributed Symbolic Reachability Analysis
### PGAS: Memory Model

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>\cdots</th>
<th>Thread ( n - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S[0] )</td>
<td>( S[k - 1] )</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
</tr>
<tr>
<td>( S[k] )</td>
<td>( S[2k - 1] )</td>
<td>( \cdots )</td>
<td>( S[(n - 1)k] )</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
<td>( S[nk - 1] )</td>
</tr>
</tbody>
</table>

- **Shared**
- **Private**
**PGAS: memget and memput**

**memget(P, S)**
- Copies block of *shared* memory *S* into *private* memory *P*

**memput(S, P)**
- Copies block of *private* *P* memory into *shared* memory *S*
PGAS: Synchronous Operations
PGAS: Asynchronous Operations

Initiator

RDMA device of initiator

RDMA device of target

Main-memory of target

Continue work...
Many Implementations:

- Berkeley UPC
- OpenSHMEM
- Co-array Fortran
- Titanium
- X10
- Chapel
- etc.

We chose UPC because it supports async operations
For messages \( \leq 16 \) bytes:

- Local 76 times faster than remote
- Local: 50\(\text{ns}\) on average
- Remote: 3.87\(\mu\text{s}\) on average
For messages ≤ 16 bytes:

- Local only 6 times faster than remote
- Local: 0.44\(\mu s\) on average
- Remote: 2.68\(\mu s\) on average
For messages \( \leq 16 \) bytes:

- Local throughput 48 times higher than remote throughput
1 Introduction
2 Preliminaries on Infiniband & RDMA
3 Part 1: Storing State Spaces
4 Part 2: Load-Balancing
5 Part 3: Distributed BDD Operations
6 Experimental Evaluation
7 Conclusion
Hash Table: Notations

**Hash Table**

\[ T = \langle b_0, \ldots, b_{n-1} \rangle \]

as a sequence of buckets \( b_i \), where:

- Hash table size: \( n \), number of inserted elements: \( m \)
- Load factor: \( \alpha = \frac{m}{n} \)

**Hash function**

\( h : U \rightarrow R \), with:

- Range of keys: \( R = \{0, \ldots, r - 1\} \)
- Universe: \( \{0, 1\}^w \) *(of all \( w \)-sized binary words)*
- Mapping words \( x \in U \) to buckets \( b_{h(x)} \) by letting \( r < n \)

**Notation**

- For \( x \in U \), we write \( x \in b_i \) if bucket \( b_i \) contains \( x \)
- For \( x \in U \), we write \( x \in T \) if \( x \in b_i \) for some \( 0 \leq i \leq n - 1 \)
Distributed Hash Table: Requirements

**Requirements**

1. Minimal number of roundtrips
2. Minimal memory overhead
3. CPU efficient (no polling)
4. Should support `find-or-put`
5. Should support PGAS

**find-or-put(d)**

- If $d \in T$, return `found`
- If $d \notin T$, insert $d$ and return `inserted`
- If $d \notin T$ and $d$ cannot be inserted, return `full`
Resolving Collisions

Hash collision

Occurs when \( h(x) = h(y) \) for \( x, y \in U \) with \( x \neq y \)
- Hashing strategy determines the number of roundtrips required by find-or-put

Existing work (RDMA-based key/value stores)
- Pilaf (Cuckoo hashing)
- Nessie (Cuckoo hashing)
- FaRM (Hopscotch hashing)
- HERD (high throughput, but CPU inefficient)

We investigated hashing strategies and determined their performance
Chained Hashing

- Every bucket is a linked list
- Inserting $x \in U$ performed by adding it to $b_{h(x)}$
- Finding $x \in U$ performed by traversing $b_{h(x)}$

Complexity of find-or-put($d$)

- $\Theta(m)$ in worst case (when all $m$ elements are in $b_{h(d)}$)
- $\Theta(1 + \alpha)$ on average when a universal hash function is used

Universal hash function

$h : U \rightarrow R$ is called universal if $Pr[h(x) = h(y)] \leq \frac{1}{|U|}$ for every $x, y \in U$

- Good theoretical properties
- In practice: "cheaper" functions are often used
Cuckoo Hashing

Cuckoo Hashing

- Uses $k \geq 2$ independent hash functions $h_1, \ldots, h_k : U \rightarrow R$ with $h_i \neq h_j$ for every $i \neq j$ (k-way Cuckoo hashing)
- Nessie: 2-way Cuckoo hashing
- Pilaf: 3-way Cuckoo hashing

Cuckoo Invariant

For every element $x \in U$ it holds that either $x \notin T$ or $x \in b_{h_i(x)}$ for exactly one $1 \leq i \leq k$

Complexity

- Lookups require $k$ roundtrips
- Inserts may require many when all $k$ buckets are occupied
Bucketized Cuckoo Hashing

- Every bucket $b_i$ is subdivided into $l$ slots
- Every slot may contain an element from $U$
- Denoted by $(k, l)$-Cuckoo hashing

Bucketized Cuckoo Hashing

- Same as Cuckoo hashing, but linearly reduced by $l$
- Efficient even when $\alpha > 0.9$ (Andersen et al, 2013)
- Pilaf: $(2, 4)$-Cuckoo hashing could be very effective
Each bucket $b_i$ has a fixed-sized neighborhood $N(b_i)$ of constant size $H \geq 1$.

$N(b_i) = \langle b_i, \ldots, b_j \rangle$ with $j = (i + H - 1) \mod n$.

$N(b_i)$ thus contains $b_i$ itself and the next $H - 1$ buckets (modulo $n$).

Neighborhoods are thus consecutive in memory.

Hopscotch Invariant

Let $x \in U$ and $N(b_{h(x)}) = \langle b_1, \ldots, b_H \rangle$. Then either $x \notin T$ or $x \in b_i$ for exactly one $1 \leq i \leq H$.

Complexity

- `find-or-put(d)` may obtain $N(b_{h(d)})$ in 1 roundtrip.
- Inserts may require many more when $N(b_{h(d)})$ is full.
Linear Probing

For $x \in U$, it examines $b_h(x)+0, b_h(x)+1, \ldots, b_h(x)+t \pmod{n}$ with threshold $t > 0$

- Buckets are consecutive in memory
- Therefore, cache-line efficient

Complexity

- Same as Hopscotch, but without relocation schemes
- Hopscotch invariant not maintained, lookups are more expensive
- But inserts are arguable cheaper \textit{(amortized complexity)}
Complexity of Linear Probing

**Theorem: Examining buckets (Knuth, 1997)**

Assuming that a *universal hash function* is used, the expected number of buckets to examine until an empty bucket is found is at most:

\[
\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)
\]

**Chunk retrieval**

- Similar to Hopscotch, a fixed-sized range of buckets can be obtained with a *single* roundtrip, which we refer to as *chunks*
- We denote the *chunk size* by \( C \geq 1 \)
Complexity of Chunk Retrievals

**Corollary: Number of chunks**

The expected number of chunks to be inspected is at most:

$$\frac{1}{2C} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$

**Number of chunks to read**

![Graph showing the number of chunks to read as a function of the load-factor \(\alpha\) for different values of \(C\).](image)
Theorem: Efficiency bound

A chunk of size $C \geq 1$ is expected to contain an empty bucket if:

$$\alpha \leq 1 - \sqrt{\frac{1}{2C - 1}}$$

Expected load-factor at which chunk is full
Designing find-or-put

Memory layout

- Shared array $B[0], \ldots, B[kn - 1]$ of buckets, so that each thread owns $k$ buckets
- 2D array $P[0][0], \ldots, P[M - 1][C - 1]$ per thread in private memory

Bucket layout

| Bit offset | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| #          |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|            | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| Bit number |
|            |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

(1) is a locking bit (1 bit)
(2) contains data (63 bits)
Cache Efficiency

Cache lines
- Typically 64 bytes in size
- So 8 buckets per cache line
- Therefore, we choose $C$ to be a multiple of 8

Cache line alignment
- The arrays $P$ are cache line aligned
- The array $B$ is *not*, since it is shared (could not find support from UPC to align shared memory)
- But the IB verbs library *has* support for shared memory alignment...
**Asynchronous chunk retrievals**
- Before iterating over a chunk, request the *next* consecutive chunk
- Done to overlap roundtrips with actual work (*interleaving queries*)

**The query-chunk\((i, p)\) operation**
- Transfers the *\(i\)*th chunk, starting from \(b_p\), from \(B\) into \(P[i][0], \ldots, P[i][C - 1]\) *asynchronously*
- Returns a *handle* \(r\) for synchronization
- Synchronization can be performed by calling \(\text{sync}(r)\)
Design of find-or-put

```python
def find-or-put(data):
    h ← hash(data)
    s_0 ← query-chunk(0, h)
    for i ← 0 to M - 1:
        if i < M - 1
            s_{i+1} ← query-chunk(i + 1, h)
            sync(s_i)
        for j ← 0 to C - 1:
            if (P[i][j] & OCCUPIED) = 0
                a ← (h + iC + j) mod kn
                d ← data(data) | OCCUPIED
                val ← cas(B[a], P[i][j], d)
                if val = P[i][j]
                    return inserted
            elif data(val) = data
                return found
            elif data(P[i][j]) = data
                return found
    return full
```

```python
def query-chunk(i, h):
    ▶ Find start and end index
    start ← (h + iC) mod kn
    end ← (h + (i + 1)C - 1) mod kn
    if end < start
        return split(start, end)
    else
        S ← ⟨B[start], . . . , B[end]⟩
        P ← ⟨P[i][0], . . . , P[i][C - 1]⟩
        return memget-async(S, P)
```

```python
def split(start, end):
    ▶ Find the blocks in shared memory
    S_1 ← ⟨B[start], . . . , B[kn - 1]⟩
    S_2 ← ⟨B[0], . . . , B[end]⟩
    ▶ Corresp. blocks in private memory
    P_1 ← ⟨P[i][0], . . . , P[i][|S_1| - 1]⟩
    P_2 ← ⟨P[i][|S_1|], . . . , P[i][C - 1]⟩
    ▶ Retrieve the chunk
    s_1 ← memget-async(S_1, P_1)
    s_2 ← memget-async(S_2, P_2)
    return ⟨s_1, s_2⟩
```
Asynchronous Query Retrievals

\[ s_0 = \text{query-chunk}(0, d) \]
\[ s_1 = \text{query-chunk}(1, d) \]
\[ s_2 = \text{query-chunk}(2, d) \]
\[ s_3 = \text{query-chunk}(3, d) \]
\[ s_4 = \text{query-chunk}(4, d) \]

\[ \text{sync}(s_0) \]
\[ \text{sync}(s_1) \]
\[ \text{sync}(s_2) \]
\[ \text{sync}(s_3) \]

\[ \cdots \]
Experimental Evaluation

Experimental setup (m610 partition)

- 10 Dell M610 machines
- 8 GPU cores and 24 GB main-memory (each)
- Ubuntu 14.04.2 LTS, kernel version 3.13.0
- 20 GB/s Infiniband network

Benchmarks

- Throughput of find-or-put
- Latency of find-or-put
- Roundtrips required by find-or-put
- Under different workloads: mixed, read-intensive, and write-intensive
Local Throughput of find-or-put

TP per thread (left) and total TP (right)

### Workload

<table>
<thead>
<tr>
<th>Workload</th>
<th>Base Throughput</th>
<th>Best Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Throughput</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Procs.</td>
<td>Procs.</td>
<td></td>
</tr>
<tr>
<td>Mixed</td>
<td>324,676,333</td>
<td>2,049,388,900</td>
<td>6.31</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Read-intensive</td>
<td>376,434,000</td>
<td>2,342,278,333</td>
<td>6.22</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Write-intensive</td>
<td>422,593,000</td>
<td>1,963,570,267</td>
<td>4.65</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Remote Throughput of find-or-put

Mixed workload: TP per thread (left) and total TP (right)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Base Throughput</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP.</td>
<td>M.</td>
<td>Procs./M.</td>
</tr>
<tr>
<td>Mixed</td>
<td>592,929</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Read</td>
<td>742,728</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Write</td>
<td>495,370</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Best Throughput</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TP.</td>
<td>M.</td>
<td>Procs./M.</td>
</tr>
<tr>
<td>Mixed</td>
<td>3,607,003</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Read</td>
<td>4,620,752</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Write</td>
<td>2,999,234</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed</td>
<td>6.08</td>
</tr>
<tr>
<td>Read</td>
<td>6.22</td>
</tr>
<tr>
<td>Write</td>
<td>6.05</td>
</tr>
</tbody>
</table>
Speedups in Remote Throughput

- Mixed
- Read-intensive
- Write-intensive
Latency of find-or-put

Local latency

![Local latency graph]

Remote latency

![Remote latency graph]
Roundtrips Required by \texttt{find-or-put}

Number of roundtrips

\begin{itemize}
\item \texttt{find-or-put} requires 0.4 roundtrips at a load-factor of 0.2.
\item \texttt{find-or-put} requires 0.8 roundtrips at a load-factor of 0.8.
\item \texttt{find-or-put} requires 1.2 roundtrips at a load-factor of 0.85.
\item \texttt{find-or-put} requires 1.6 roundtrips at a load-factor of 0.9.
\end{itemize}

\begin{itemize}
\item \texttt{find-or-put} requires 2 roundtrips at a load-factor of 0.2.
\item \texttt{find-or-put} requires 2.8 roundtrips at a load-factor of 0.8.
\item \texttt{find-or-put} requires 4 roundtrips at a load-factor of 0.85.
\item \texttt{find-or-put} requires 6 roundtrips at a load-factor of 0.9.
\end{itemize}

\textbf{Load-factor ($\alpha$)}

\textbf{Roundtrips}

\begin{itemize}
\item $C = 8$
\item $C = 16$
\item $C = 32$
\item $C = 64$
\item $C = 128$
\end{itemize}
Conclusions

General
- Minimizing roundtrips *critical* for increased performance
- Overlapping queries reduces waiting-times and increases latency
- Linear probing requires less roundtrips than Hopscotch and Cuckoo

Performance
- find-or-put takes 9.3\(\mu\)s on average with \(\alpha = 0.9\) and \(C = 64\)
- Peak-throughput of \(3.6 \times 10^6\) op/s obtained with 10 machines

Future work
- Use adaptive chunk sizes (based on efficiency bounds Theorem)
- In addition, update asynchronous queries to prevent unused retrievals
1 Introduction
2 Preliminaries on Infiniband & RDMA
3 Part 1: Storing State Spaces
4 Part 2: Load-Balancing
5 Part 3: Distributed BDD Operations
6 Experimental Evaluation
7 Conclusion
Load Balancing

Task-based parallelism

- Dividing computational problems into smaller *tasks*
- Task is a basic unit of work and only depend on intermediate *subtasks*
- All threads maintain *task pools*

Load-balancing tasks

- Ideally tasks are perfectly distributed (*infeasible*)
- Instead: mapping tasks dynamically to threads

Task granularity

The relation between the computational workload and the amount of communication required between threads

- Fine-grained: large number of small tasks
- Coarse-grained: small number of large tasks
Work Stealing and Sharing

### Work stealing

- Efficient technique for fine-grained task parallelism
- Threads are either idle or working
- When idle, threads steal from remote task pools
- Stealing thread is thief, targetted thread is victim
- Termination when all threads are idle

### Work sharing

- Threads communicate their status
- When idle, other threads share work
- Communication of work stealing is more efficient (Blumofe, 1999)
Work Stealing Operations

Operations

- **spawn**: push new task to task pool
- **call**: execute given task
- **sync**: pull task from pool and execute

Fibonacci example

```c
1 int fib(n):
  2 if n < 2 return n
  3 a ← fib(n - 1)
  4 b ← fib(n - 2)
  5 return a + b
```

```c
1 int par-fib(n):
  2 if n < 2 return n
  3 spawn(par-fib, n - 1)
  4 r ← call(par-fib, n - 2)
  5 return r + sync
```
Implementing the Task Pool

### Double ended queue (deque)
- Similar to queue, but has two ends: head and tail
- Items can be pushed or popped from both ends
- Implemented as a fixed-sized array

#### Example

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Initial

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- push(3, h)

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- pop(h)
Implementing the Task Pool

**Split deque**
- Deque with a *split point* $s$
- $s$ determines what sections belong to *head* and *tail*
- Used to denote a *public* and *private* region
- $s$ can be relocated to increase/decrease public region

**Example**

```
Initial

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

push(8, t)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

push(3, h)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

pop(h)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Implementing the Task Pool

Performance of split deques
- Modifying s may conflict with steal operations
- Either locks or memory fences required
- Expensive in distributed environment!

Existing work (current state-of-the-art)
- HotSLAW: access to public region requires locking
- Scioto: whole split deque locked when stealing
- Lace: non-blocking, but shrinking public region requires memory fence
Implementing the Task Pool

Private deques
- Implemented as a stack
- Do not have a public region (completely private)

Private deque work stealing
- When stealing, idle workers explicitly ask for work
- **Advantage:** No locking required
- **Disadvantage:** Requires participation from both victim and thief
Victim Selection Protocols

**Selecting victims**
- Random victim selection
- Hierarchical victim selection (Min et al, 2011)
- Leapfrogging

**Contribution and motivation**
Private-deque work stealing operations:
- Minimal number of roundtrips
- Uses all three victim selection protocols
- Similar approach by Olivier et al. (2008), but requires *more* roundtrips and does *not* exploit network hierarchy
Designing Private-Deque Work Stealing

Memory layout

- Shared 2D-array: deque[0][0], ..., deque[THREADS - 1][k - 1]
- Request cells: request[0], ..., deque[THREADS - 1]
- Transfer cells: transfer[0], ..., transfer[THREADS - 1]
- Status cells: status[0], ..., status[THREADS - 1]

Schematically

```
private deque                 request      transfer      status
[ ] [ ] [ ] ... [ ] [ ] [ ]
           (k x m)B          8B           (8 + m)B       4B
```

Where $m$ is the task size (in Bytes) and $k$ the deque size
Performing Steals

**Request cell**
- Contains either **BLOCKING**, **EMPTY**, or a thread ID:
  - **blocking**: no tasks can be stolen
  - **empty**: no pending steal requests
  - **identifier**: pending steal request

**Transfer cell**
- Contains either **EMPTY** or a task + location:
  - **empty**: no task received
  - **task**: task received + corresponding location in deque

**Status cell**
- Contains either **IDLE** or **WORKING**
Hierarchical Work Stealing

Domain levels

- Berkeley UPC provides \texttt{thread-distance}(i, j) function
- Which returns: verynear, near, far, or veryfar
- We use an array \textit{domain}, so that \textit{domain}[i] contain all thread IDs on the \textit{i}th level
- We use a \texttt{shuffle} function that randomly \textit{shuffles} a domain level.

Hierarchical stealing

1. Threads start by performing leapfrogging
2. Threads perform \texttt{count}([\textit{domain}[i]]) steal attempts before moving to level \textit{i} + 1
3. If all levels have been tried, perform termination detection
Designing \texttt{spawn}, \texttt{call}, and \texttt{sync}

\begin{verbatim}
1 def sync():
2     task ← deque[MY-ID][head − 1]
3     if task.stolen
4         communicate()
5     while ¬task.completed:
6         ▶ Perform leapfrogging
7             if steal(task.owner) continue
8             if task.completed break
9         ▶ Perform hierarchical stealing
10        for i ← 0 to HIERARCHY-LVLS - 1:
11            shuffle(domain[i])
12            foreach victim ∈ domain[i] do
13                if steal(victim) goto line 5
14                if task.completed goto line 16
15        ▶ Return result from stolen task
16    head ← head − 1
17    tail ← tail − 1
18    return task.result
19 else
20    head ← head − 1
21    return call(task)
\end{verbatim}

\begin{verbatim}
1 def spawn(desc, params):
2     ▶ Build a new task
3         task ← deque[MY-ID][head]
4         task.desc ← desc
5         task.stolen ← false
6         task.completed ← false
7         task.params ← params
8     ▶ Write new task to deque
9         deque[MY-ID][head] ← task
10         head ← head + 1
11 def call(desc, params):
12     ▶ Find the intended function
13         func ← function-of(desc)
14     ▶ Invoke that function
15     return func(params)
\end{verbatim}
def initiate(desc, params):
    # Wait for all workers to start
    status[MY-ID] ← WORKING
    barrier()
    # Perform task
    result ← call(desc, params)
    # Wait for all workers to complete
    status[MY-ID] ← IDLE
    barrier()
    return result

def compute(desc, params):
    if MY-ID = 0
        initiate(desc, params)
    else
        participate()

def participate():
    # Wait for all workers to start
    status[MY-ID] ← IDLE
    barrier()
    # Perform hierarchical stealing
    while true:
        status[MY-ID] ← IDLE
        for i ← 0 to HIERARCHY-LVLS - 1:
            shuffle(domain[i])
            foreach victim ∈ domain[i] do
                if steal(victim) goto line 5
        # No worker had tasks to steal..
        if termination-detection() break
        barrier()
```python
def communicate():
    if head - tail < 2
        if request[MY-ID] ≠ BLOCKED
            △ Not enough stealable tasks, block further requests
        if request[MY-ID] ≠ EMPTY
            reject-and-block()
        elif cas(request[MY-ID], EMPTY, BLOCKED) ≠ EMPTY
            reject-and-block()
        elif request[MY-ID] = BLOCKED
            request[MY-ID] ← EMPTY
        elif request[MY-ID] ≠ EMPTY
            thief ← request[MY-ID]
            request[MY-ID] ← EMPTY
            △ Prepare task to be stolen
deque[MY-ID][tail].stolen ← true
deque[MY-ID][tail].owner ← thief
            △ Construct the transfer message
            msg ← new TransferMessage
            msg.index ← tail
            msg.task ← deque[MY-ID][tail]
            memput-async(transfer[thief], msg)
```
def steal(victim):
    communicate()
    transfer[MY-ID] ← EMPTY
    res ← cas(request[victim], EMPTY, MY-ID)
    if res = EMPTY
        ▷ Wait for response from victim
        while transfer[MY-ID] = EMPTY:
            communicate()
            if transfer[MY-ID] = EMPTY
                return false
    else
        status[MY-ID] ← WORKING
        i ← transfer[MY-ID].index
        task ← transfer[MY-ID].task
        task.result ← call(task)
        ▷ Write back the task result
        task.completed ← true
        memput-async(deque[victim][i], task)
    return true

def reject-and-block():
    ▷ Block further requests
    thief ← request[MY-ID]
    request[MY-ID] ← BLOCKED
    ▷ Send a negative response
    msg ← new TransferMessage
    msg.index ← 0
    msg.task ← EMPTY
    memput-async(transfer[thief], msg)

def termination-detection():
    ▷ Send status requests
    for i ← 0 to THREADS − 1:
        s_i ← memget-async(res[i], status[i])
    ▷ Wait for status responses
    for i ← 0 to THREADS − 1:
        sync(s_i)
        if res[i] = WORKING
            return false
    return true
Experimental Evaluation

Benchmarks

- Performed a number of microbenchmarks
- Determined speedup when scaling along machines and threads per machine
- Compared speedup with HotSLAW

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nr. of Tasks</th>
<th>Avg. Task Time</th>
<th>Input/Output Size</th>
<th>Input/Output Size Hotslaw</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(45)</td>
<td>3,672,623,805</td>
<td>0.154 μs</td>
<td>16/8 bytes</td>
<td>4/8 bytes</td>
</tr>
<tr>
<td>nqueens(15)</td>
<td>171,127,071</td>
<td>4.14 μs</td>
<td>20/8 bytes</td>
<td>28/8 bytes</td>
</tr>
<tr>
<td></td>
<td>96,793,510</td>
<td>0.986 μs</td>
<td>20/8 bytes</td>
<td>32/0 bytes</td>
</tr>
<tr>
<td></td>
<td>111,345,631</td>
<td>0.722 μs</td>
<td>20/8 bytes</td>
<td>32/0 bytes</td>
</tr>
<tr>
<td></td>
<td>32,767</td>
<td>188.30 μs</td>
<td>20/8 bytes</td>
<td>28/0 bytes</td>
</tr>
<tr>
<td>uts(T2L)</td>
<td>96,793,510</td>
<td>0.986 μs</td>
<td>20/8 bytes</td>
<td>32/0 bytes</td>
</tr>
<tr>
<td>uts(T3L)</td>
<td>111,345,631</td>
<td>0.722 μs</td>
<td>20/8 bytes</td>
<td>32/0 bytes</td>
</tr>
<tr>
<td>matmul(512)</td>
<td>32,767</td>
<td>188.30 μs</td>
<td>20/8 bytes</td>
<td>28/0 bytes</td>
</tr>
</tbody>
</table>
Speedup Graphs

fib(45)  
matmul(512)  
uts(T3L)  
nqueens(15)  
uts(T2L)
## Computation Times

### Sequential- and best times

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Sequential Time</th>
<th>Best Configuration</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Machines</td>
</tr>
<tr>
<td>fib(45)</td>
<td>563.87</td>
<td>8.85</td>
<td>10</td>
</tr>
<tr>
<td>matmul(512)</td>
<td>6.17</td>
<td>1.07</td>
<td>1</td>
</tr>
<tr>
<td>uts(T2L)</td>
<td>90.48</td>
<td>1.90</td>
<td>10</td>
</tr>
<tr>
<td>uts(T3L)</td>
<td>73.60</td>
<td>3.48</td>
<td>10</td>
</tr>
<tr>
<td>nqueens(15)</td>
<td>707.64</td>
<td>10.29</td>
<td>10</td>
</tr>
</tbody>
</table>

### Comparison with HotSLAW

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Our Implementation</th>
<th>HotSLAW</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seq. Time</td>
<td>Best Time</td>
<td>Seq. Time</td>
</tr>
<tr>
<td>fib(45)</td>
<td>563.87</td>
<td>8.85</td>
<td>938.49</td>
</tr>
<tr>
<td>nqueens(15)</td>
<td>707.64</td>
<td>10.29</td>
<td>387.53</td>
</tr>
<tr>
<td>uts(T2L)</td>
<td>90.48</td>
<td>1.90</td>
<td>81.22</td>
</tr>
<tr>
<td>uts(T3L)</td>
<td>73.60</td>
<td>3.48</td>
<td>67.64</td>
</tr>
<tr>
<td>uts(T3L)*</td>
<td>73.60</td>
<td>3.48</td>
<td>51.69</td>
</tr>
</tbody>
</table>
Comparison with HotSLAW (Computation Time)

- fib(45)
- nqueens(15)
- uts(T3L)
- uts(T3L) with steal-10
- uts(T2L)
Average Number of Steals

- **fib(45)**
- **nqueens(15)**
- **uts(T3L)**
- **uts(T3L) with steal-10**
- **uts(T2L)**
Latency of steal

local

remote

HotSLAW - local

HotSLAW - remote
Speedup of steal (vs HotSLAW)

local

remote