Background

- PhD at Formal Methods & Tools, University of Twente
- PhD Research: Parallel Binary Decision Diagrams
  - Using work-stealing...
  - ...and lock-free hash tables
  - to implement Sylvan and Lace.
- Current research interests
  - Parallel SAT
  - Parity games
What to do as a good student?

- I want you to understand each slide.
- Ask me why I made certain choices.
- Ask me how to find performance problems.
- Ask me how to fine-tune the implementation.
- Ask me about the relation between shared-memory and message passing.
- Ask me why I think we cannot go much faster than this.
Task parallelism

def fib(n):
    if n < 2: return n
    spawn fib(n - 1)
y = fib(n - 2)
x = sync
return x + y

Lace: non-blocking split deque for work-stealing

Tom van Dijk & Jaco van de Pol
Example: calculate $\text{fib}(11)$

Task graph:

```
          11
         / \  \\
       10   9
      /   /  \\
     8   7   \\
    /   /    \\
   6   5    \\
  /   /     \\
 4   3      \\
/   /       \\
2   1       \\
```

Task stack:

```
10  8  6  4  2
```
Deques for work-stealing

Deques

A **deque** is a double-ended queue.
Every worker has a deque to store tasks in.
Workers steal tasks from each other when they have no work.

- **push**: add a task at the head (end) of the deque
- **pop**: remove the task at the head of the deque
- **steal**: steal the task at the tail of the deque

Implementations (**blue** = non-blocking)

- Private deque: Acar ea (**2013**)
Deques for work-stealing

Challenges

- Avoid hidden and unnecessary communication
  - false sharing (variables accessed by thieves / owner)
  - unnecessary memory writes
- Avoid using locks/mutexes
- Avoid overhead, especially if most tasks are never stolen
- Disadvantages of shared deques [Acar et al, PPoPP 2013]
  - Difficult to support strategies such as steal-multiple
  - Require expensive memory fences (in every pop)
  
  For example, in the THE protocol of Cilk
Implemented non-blocking split deque in Lace. Deque is described by variables tail ($t$), split ($s$), head ($h$).

<table>
<thead>
<tr>
<th>$t$</th>
<th>$s$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- Everything before $s$ is **shared**.
- Everything before $t$ is **stolen**.
- **steal:**
  - if $t < s$: steal with atomic `cas $\langle t, s \rangle \rightarrow \langle t + 1, s \rangle`  
  - if $t \geq s$: set flag `splitreq`
- Thieves can only increase $t$, not modify $s$ or $h$.
- Thieves access $t$ and $s$ on a separate cacheline from owner.
- Stolen tasks stay in the deque!
Deque in Lace

Push/pop a task

- **push**: write task at $h$ and increase $h$.
- **pop**: if $h > s$, return task at $h$ and decrease $h$.
Deque in Lace

Grow and shrink

- **grow**: set $s$ between $t$ and $h$ (round up).

<table>
<thead>
<tr>
<th>$ts$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

- **shrink**: set $s$ between $t$ and $h - 1$ (round up).

<table>
<thead>
<tr>
<th>$t$</th>
<th>$sh$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$ts$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Deque in Lace

Shrink conflict (x86 load-before-store)

- **steal during shrink on tasks beyond new s**
- **solution: wait in memory fence and check t**

```
  t  sh
  10 8  6  4
```

```
  s  t  h
  10 8  6  4
```

```
  ts  h
  10 8  6  4
```

```
  tsh
  10 8  6
```
Private deque by Acar et al (2013)

- Also implemented in Lace framework for comparison.
- Every worker has a request cell $r$ and a transfer cell $t$.
- A thief writes atomically ($\text{cas}$) to $r$ of a victim and waits.
- The victim writes result in $t$ of the thief.
- Workers must regularly check $r$ to communicate tasks.
Experimental results

Benchmarks

- \texttt{fib(50)} – 20,365,011,073 tasks
- \texttt{N-queens(15)} – 171,129,071 tasks
- \texttt{uts(T3L)} – Unbalanced Tree Search, 111,345,630 tasks
- \texttt{matmul(4096)} – 3,595,117 tasks
- No cut-off point
- Fine-grained, very small tasks.

Measurements

- Compare Lace split deque to private deque (and to Wool)
- 48-core AMD machine (4 sockets, 12 cores per socket)
- Wallclock time around parallel part, 48 workers.
Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$T_S$</th>
<th>Lace</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_{48}$</td>
<td>$T_S / T_{48}$</td>
</tr>
<tr>
<td>fib 50</td>
<td>149.2</td>
<td>144</td>
<td>4.13</td>
</tr>
<tr>
<td>uts T3L</td>
<td>43.11</td>
<td>44.2</td>
<td>2.23</td>
</tr>
<tr>
<td>uts T3L *</td>
<td>43.11</td>
<td>44.3</td>
<td>1.154</td>
</tr>
<tr>
<td>queens 15</td>
<td>533</td>
<td>602</td>
<td>12.63</td>
</tr>
<tr>
<td>matmul 4096</td>
<td>773</td>
<td>781</td>
<td>16.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private deque</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fib 50</td>
<td>149.2</td>
<td>208</td>
<td>5.22</td>
</tr>
<tr>
<td>uts T3L</td>
<td>43.11</td>
<td>44.8</td>
<td>2.55</td>
</tr>
<tr>
<td>uts T3L *</td>
<td>43.11</td>
<td>44.8</td>
<td>1.240</td>
</tr>
<tr>
<td>queens 15</td>
<td>533</td>
<td>541</td>
<td>11.34</td>
</tr>
<tr>
<td>matmul 4096</td>
<td>773</td>
<td>774</td>
<td>16.34</td>
</tr>
</tbody>
</table>

* = with extension to fix issues with leapfrogging (next slides)
Scalability $T_1/T_n$ (relative to itself; does not show overhead)
Scalability $T_S/T_n$ (relative to sequential; shows overhead)

![Scalability Graph](image)

Name
- fib
- fib-ri
- uts-t3l
- uts-t3l-ri

Tom van Dijk & Jaco van de Pol

Lace: non-blocking split deque for work-stealing
Leapfrogging

- Waiting for stolen work? Steal from thief!
- Advantage: gives nice upper bound on deque size!
- Disadvantage: steal chaining...

- New tasks by w11 are stolen by w10...
- New tasks by w10 are then stolen by w11... and w9...
- New tasks by w9 are then stolen by w10... and w8...
- Work does not trickle down fast enough!
Leapfrogging

- Leapfrogging results in steal chaining.

Transitive Leapfrogging (by Faxén)

- If thief has no work, steal from thief of thief.
- Implemented in Wool, works well!
- Disadvantage: requires more communication.

Leapfrogging into random stealing

- If thief has no work, steal from random target.
- Very simple and works well!
- Disadvantage: no guarantee on deque size upper bound.
## Leapfrogging

### Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$T_1$</th>
<th>$T_{48}$</th>
<th>$T_{S}/T_{48}$</th>
<th>$T_1/T_{48}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Lace</td>
<td>44.2</td>
<td>2.23</td>
<td>18.7</td>
<td>19.9</td>
</tr>
<tr>
<td>Old Private Deque</td>
<td>44.8</td>
<td>2.55</td>
<td>17.3</td>
<td>17.5</td>
</tr>
<tr>
<td>Old Wool</td>
<td>44.3</td>
<td>2.12</td>
<td>19.4</td>
<td>20.9</td>
</tr>
<tr>
<td>Lace</td>
<td>44.26</td>
<td>1.154</td>
<td>37.4</td>
<td>38.3</td>
</tr>
<tr>
<td>Private Deque</td>
<td>44.83</td>
<td>1.240</td>
<td>34.8</td>
<td>36.2</td>
</tr>
<tr>
<td>Wool</td>
<td>44.27</td>
<td>1.172</td>
<td>36.8</td>
<td>37.8</td>
</tr>
</tbody>
</table>

- All algorithms similar speedup
- Peak stack depth from 6500-12500 tasks to 17000-21000 tasks (1 MB)
Conclusions

- Non-blocking split deque has low overhead and good speedup
- Leapfrogging plus random stealing solves steal chaining
- Only require memory fence in `shrink`
- Lace can be found at:
  - http://github.com/trolando/lace
  - Feel free to reproduce results (bench.py)
- Lace is used in our parallel BDD implementation Sylvan
Conclusions

- Non-blocking split deque has low overhead and good speedup
- Leapfrogging plus random stealing solves steal chaining
- Only require memory fence in `shrink`
- Lace can be found at:
  - http://github.com/trolando/lace
  - Feel free to reproduce results (bench.py)
- Lace is used in our parallel BDD implementation Sylvan

Future directions

- Distributed memory (with a shared memory abstraction)
- Non-uniform task size
def steal():
    if allstolen: return None
    (t, s) = (tail, split)
    if t < s:
        if cas((tail, split), (t, s), (t+1, s)):
            return Task(t)
        else: return None
    if ! splitreq: splitreq=1
    return None

def push(data):
    if head == size: return FULL
    write task data at head
    head = head + 1
    if o_allstolen:
        (tail, split) = (head-1, head)
        allstolen = 0
        if splitreq: splitreq=0
        o_split = head
        o_allstolen = 0
    elif splitreq: grow_shared()
def pop():
    if head = 0: return EMPTY,
    if o_allstolen: return STOLEN, Task(head-1)
    if o_split = head:
        if shrink_shared(): return STOLEN, Task(head-1)
    head = head-1
    if splitreq: grow_shared()
    return WORK, Task(head)

def pop_stolen():
    head = head-1
    if ! o_allstolen:
        allstolen = 1
        o_allstolen = 1
def grow_shared():
    new_s = (o_split+head+1)/2
    split = new_s
    o_split = new_s
    splitreq = 0

def shrink_shared():
    (t,s) = (tail,split)
    if t != s:
        new_s = (t+s)/2
        split = new_s
        o_split = new_s
        MFENCE
        t = tail # read again
    if t != s:
        if t > new_s:
            new_s = (t+s)/2
            split = new_s
            o_split = new_s
            return False
    allstolen = 1
    o_allstolen = 1
    return True