Lace: Non-Blocking Split Deque for Work-Stealing
Tom van Dijk & Jaco van de Pol
FMV/Parallel Computing, a sunny morning in 2017
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 Satisfiability
  - Using ZBDDs to store clause sets for Satisfiability
  - Solving Parity Games via Priority Promotion
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

```python
def fib(k):
    if k < 2: return k
    spawn fib(k - 2)
    spawn fib(k - 1)
    n ← sync
    m ← sync
    return n + m
```
Example: calculate $\text{fib}(11)$

Task graph:

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

Task deque (of first worker):

```
t                h
10 8 6 4 2
```
Work-stealing related to its deque

<table>
<thead>
<tr>
<th>Work-stealing operations</th>
<th>Deque operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>spawn(task)</td>
<td>push(task)</td>
</tr>
<tr>
<td>sync</td>
<td>peek, pop</td>
</tr>
<tr>
<td>steal-and-run(victim)</td>
<td>steal</td>
</tr>
</tbody>
</table>

- Each worker has 1 deque.
- Worker uses push/peek/pop on its own deque.
- Worker uses steal on other deques.
- Policy: steal from the thief.
Deques for work-stealing

Implementations (blue = non-blocking)

- Private deque: Acar ea (2013)
- Non-blocking split deque: Van Dijk & Van de Pol (2013)
Deques for work-stealing

Challenges

- Avoid hidden and unnecessary communication
  - false sharing (variables accessed by thieves / owner)
  - unnecessary memory writes \textit{and reads}
- Avoid using locks/mutexes
  - (solved using lock-free operations)
- Avoid expensive memory fences, e.g., Cilk-THE
  - (mostly solved using split principle)
- Avoid overhead, especially since most tasks are never stolen
  - (solved with “direct task stack”)
Deque is described by variables tail \((t)\), split \((s)\), head \((h)\).

- Tasks are **shared** or **private**.
- The first \(t\) tasks are **stolen**.
- Tasks steal by **atomic cas** on \(t\) and \(s\) together.
- Owner modifies \(h\) and \(s\) with normal memory operations.
- Extra flag: **movesplit**.
Deque in Lace

Deque is described by variables tail \((t)\), split \((s)\), head \((h)\).

\[
\begin{array}{cccc}
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Communication is key!!

<table>
<thead>
<tr>
<th>Cacheline</th>
<th>Contents</th>
<th>Thief access</th>
<th>Owner access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared 1</td>
<td>tail, split</td>
<td>Often</td>
<td>Sometimes</td>
</tr>
<tr>
<td>Shared 2</td>
<td>flag movesplit</td>
<td>Sometimes</td>
<td>Often</td>
</tr>
<tr>
<td>Private</td>
<td>head, osplit</td>
<td>–</td>
<td>Often</td>
</tr>
</tbody>
</table>
Deque in Lace

Moving the split point back

1. The owner reads $t, s$.
2. Thieves steal.
3. The owner sets $s$.
4. The owner repairs $s$.
Experimental results

Benchmarks

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

Measurements

- 48-core AMD machine (4 sockets, 12 cores per socket)
- Wallclock time around parallel part, 1, 48 workers.
# Experimental results

<table>
<thead>
<tr>
<th>Results</th>
<th>Benchmark time</th>
<th>Speedup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_S$</td>
<td>$T_1$</td>
<td>$T_{48}$</td>
</tr>
<tr>
<td>fib 50</td>
<td>149.2</td>
<td>144</td>
<td>4.13</td>
</tr>
<tr>
<td>uts T2L</td>
<td>84.5</td>
<td>86.0</td>
<td>1.81</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>44.26</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>
</tbody>
</table>

* = with extension to fix issues with leapfrogging (next slides)
Leapfrogging

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

- Work does not trickle down fast enough!
Conclusions

- Non-blocking split deque has low overhead and good speedup
- Leapfrogging plus random stealing solves steal chaining
- Only require memory fence to shrink the shared portion
- 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
Algorithm outline

```python
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
    elif ¬ movesplit:
        movesplit ← true
    return None
```

Tom van Dijk & Jaco van de Pol
Lace: non-blocking split deque for work-stealing
def push(task):
    if head == size: raise QueueFull
    write task data at head
    head ← head + 1
    if oallstolen:
        (tail,split) ← (head-1,head)
        osplit ← head
        allstolen ← false
        oallstolen ← false
        if movesplit: movesplit ← false
    elif movesplit:
        // Grow shared portion
        new_split ← (osplit + head + 1) / 2
        split ← new_split
        osplit ← new_split
        movesplit ← false
Algorithm outline

24  def pop():
25       head ← head - 1

26  def pop-stolen():
27       head ← head - 1
28       if ¬ oallstolen:
29           allstolen ← true
30           oallstolen ← true
Algorithm outline

```python
31 def peek():
32     if head==0: raise QueueEmpty
33     if oallstolen: return Stolen(head-1)
34     if osplit = head:
35         if ¬ shrink-shared():
36             allstolen ← true
37             oallstolen ← true
38             return Stolen(head-1)
39     if movesplit:
40         // Grow public section (excluding head-1)
41         new_split ← (osplit + head) / 2
42         split ← new_split
43         osplit ← new_split
44         movesplit ← false
45         return Work(head-1)
```
def shrink_shared():
    t, s ← (tail, split)
    if t = s : return false
    new_s ← (t + s) / 2
    split ← new_s
    osplit ← new_s
    memory fence
    t ← tail
    if t = s : return false
    if t > new_s :
        new_s ← (t + s) / 2
        split ← new_s
        osplit ← new_s
    return true
Informal proof

12

12

private
not stolen

14,21,41,56

shared
not stolen

14

49

5 (cas)

14,21,41,56

Stolen

Work

private
stolen

shared
stolen

14

25

27