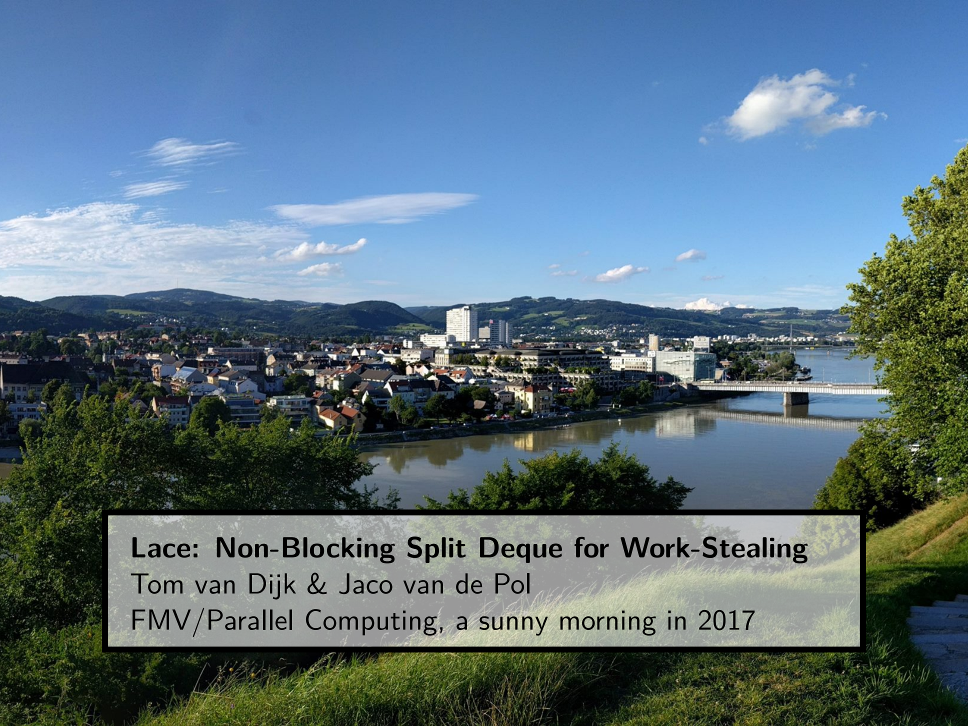


Lace: Non-Blocking Split Deque for Work-Stealing

Tom van Dijk & Jaco van de Pol

FMV/Parallel Computing, a sunny morning in 2017

Background

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

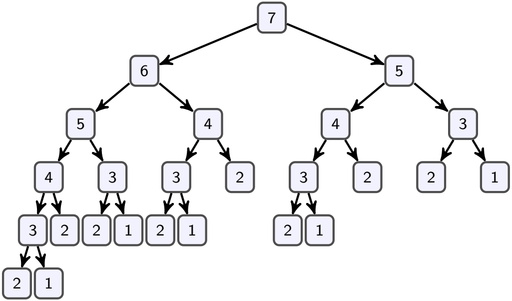
Guide

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.

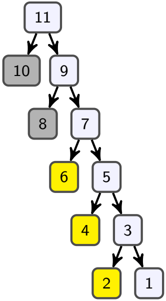
Task parallelism

```
def fib(k):
    if k < 2 : return k
    spawn fib(k − 2)
    spawn fib(k − 1)
    n ← sync
    m ← sync
    return n + m
```

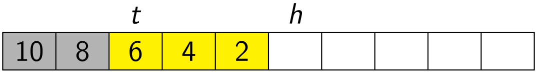


Example: calculate `fib(11)`

Task graph:



Task deque (of first worker):



Work-stealing related to its deque

Work-stealing operations	Deque operations
`spawn`(task)	`push`(task)
`sync`	`peek`, `pop`
`steal-and-run`(victim)	`steal`

- 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)

- Fully shared deque: Frigo ea ('Cilk' 1998), ABP (1998), Chase and Lev (2005), Hendler ea (2006)
- Private deque: Acar ea (2013)
- Split deque: Faxén ('Wool' 2008, 2010), Dinan ea (2009)
- Non-blocking split deque: Van Dijk & Van de Pol (2013)

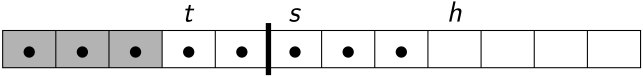
Deques for work-stealing

Challenges

- Avoid hidden and unnecessary communication
  - false sharing (variables accessed by thieves / owner)
  - unnecessary memory writes *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 in Lace

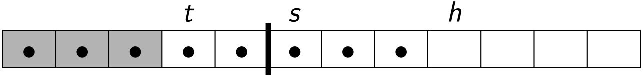
Deque is described by variables tail (t), split (s), head (h).



- 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).

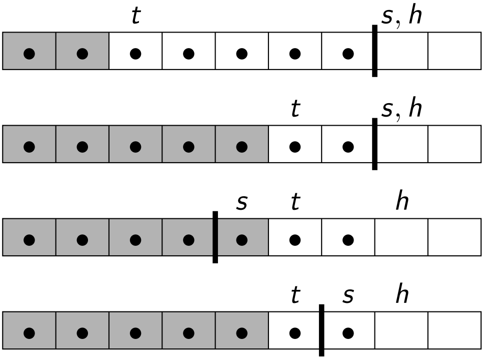


Communication is key!!

Cacheline	Contents	Thief access	Owner access
Shared 1	`tail`, `split`	Often	Sometimes
Shared 2	flag `movesplit`	Sometimes	Often
Private	`head`, `osplit`	–	Often

Deque in Lace

Moving the split point back



owner reads t,s

thieves steal

owner sets s

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.

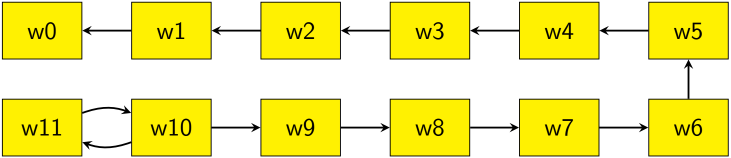
Experimental results

Results	Benchmark time			Speedup	
	T_S	T_1	T_{48}	T_S/T_{48}	T_1/T_{48}
fib `50`	149.2	144	4.13	34.5	34.9
uts `T2L`	84.5	86.0	1.81	46.1	47.4
uts `T3L`	43.11	44.2	2.23	18.7	19.9
uts `T3L` *	43.11	44.26	1.154	37.4	38.3
queens `15`	533	602	12.63	42.2	47.7
matmul `4096`	773	781	16.46	47.0	47.5

* $=$ 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

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

```
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
```

Algorithm outline

```
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

```
def pop():
    head ← head - 1
def pop-stolen():
    head ← head - 1
    if ¬ oallstolen :
        allstolen ← true
        oallstolen ← true
```

Algorithm outline

```
def peek():
    if head=0 : raise QueueEmpty
    if oallstolen : return Stolen(head-1)
    if osplit = head :
        if ¬ shrink-shared() :
            allstolen ← true
            oallstolen ← true
            return Stolen(head-1)
    if movesplit :
        // Grow public section (excluding head-1)
        new_split ← (osplit + head) / 2
        split ← new_split
        osplit ← new_split
        movesplit ← false
    return Work(head-1)
```

Algorithm outline

```
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